C++17 Standard Library Quick Reference

A Pocket Guide to Data Structures, Algorithms, and Functions

—

Second Edition

—

Peter Van Weert
Marc Gregoire

Apress®

C++17 Standard Library Quick Reference

A Pocket Guide to Data Structures, Algorithms, and Functions

Second Edition

Peter Van Weert
Marc Gregoire

Apress®

C++17 Standard Library Quick Reference: A Pocket Guide to Data Structures, Algorithms, and Functions

Peter Van Weert
Kessel-Lo, Belgium

Marc Gregoire
Meldert, Belgium

ISBN-13 (pbk): 978-1-4842-4922-2
https://doi.org/10.1007/978-1-4842-4923-9

ISBN-13 (electronic): 978-1-4842-4923-9

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484249222. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Dedicated to my parents and my brother,
who are always there for me.
Their support and patience helped me
in finishing this book.

—Marc Gregoire

In loving memory of Jeroen.
Your enthusiasm and courage will forever remain
an inspiration to us all.

—Peter Van Weert

Contents

About the Authors xv

About the Technical Reviewer xvii

Introduction xix

Chapter 1: Numerics and Math 1

Common Mathematical Functions ..... <cmath> 1

Basic Functions 1

Exponential and Logarithmic Functions 2

Power Functions 2

Trigonometric and Hyperbolic Functions 3

Integral Rounding of Floating-Point Numbers 3

Floating-Point Manipulation Functions 3

Classification and Comparison Functions 4

gcd/lcm C++17 ..... <numeric> 4

Error Handling 5

Special Mathematical Functions C++17 ..... <cmath> 5

Bessel Functions 6

Polynomials 7

Elliptic Integrals 7

Exponential Integrals 8

Error Functions 8

Gamma Functions 8

Beta Functions 9
Zeta Functions 9
Minimum, Maximum, and Clamping ..... <algorithm> 9
Fixed-Width Integer Types ..... <cstdint> 10
Arithmetic Type Properties ..... <limits> 11
Complex Numbers ..... <complex> 13
Compile-Time Rational Numbers ..... <ratio> 14
Random Numbers ..... <random> 15
Random Number Generators 15
Random Number Distributions 18
Numeric Arrays ..... <valarray> 23
std::slice 24
std::gslice 25
std::mask_array 26
std::indirect_array 27
Chapter 2: General Utilities 29
Moving, Forwarding, Swapping ..... <utility> 29
Moving 29
Forwarding 31
Swapping and Exchanging 32
Pairs and Tuples 33
Pairs ..... <utility> 33
Tuples ..... <tuple> 34
std::byte C++17 ..... <cstddef> 35
Relational Operators ..... <utility> 36
Smart Pointers ..... <memory> 36
Exclusive Ownership Pointers 36
Shared Ownership Pointers 39

Function Objects ..... <functional> 42
Reference Wrappers 43
Predefined Functors 43
Binding Function Arguments 44
Negating a Callable C++17 45
Generic Function Wrappers 45
Functors for Class Members 46
Initializer Lists ..... <initializer_list> 47
Vocabulary Types C++17 48
std::optional ..... <optional> 48
std::variant ..... <variant> 50
std::any ..... <any> 55
Date and Time Utilities ..... <chrono> 56
Durations 57
Time Points 58
Clocks 59
C-Style Date and Time Utilities ..... <ctime> 60
Type Utilities 62
Runtime Type Identification ..... <typeinfo>, <typeindex> 62
Type Traits ..... <type_traits> 63
Type Operations ..... <utility> 70
Generic Utilities 71
std::invoke C++17 ..... <functional> 71
std::addressof ..... <memory> 72

Chapter 3: Containers 73

Iterators ..... <iterator> 73

Iterator Tags 74

Non-member Functions to Get Iterators 75

Non-member Operations on Iterators 76

Sequential Containers 76

std::vector ..... <vector> 76

std::deque ..... <deque> 83

std::array ..... <array> 84

std::list and std::forward_list ..... <list>, <forward_list> 84

Sequential Containers Reference 86

std::bitset ..... <bitset> 89

Complexity 90

Reference 90

Container Adaptors 91

std::queue ..... <queue> 91

std::priority_queue ..... <queue> 91

std::stack ..... <stack> 92

Example 92

Reference 93

Ordered Associative Containers 93

std::map ..... <map> 94

Inserting in a Map 95

std::multimap ..... <map> 98

std::set and std::multiset ..... <set> 98

Order of Elements 98

Searching 99

Moving Nodes Between Containers C++17 100

Merging Containers C++17 100

Complexity 101
Reference 101
Unordered Associative Containers ..... <unordered_map>, <unordered_set> 103
Hash Map 104
Template Type Parameters 104
Hash Functions 104
Complexity 106
Reference 106
Allocators ..... <memory> 108
Polymorphic Allocators C++17 ..... <memory_resource> 108
Allocators for Multilevel Containers ..... <scoped_allocator> 111
Chapter 4: Algorithms 113
Input and Output Iterators 113
General Guidelines 114
Algorithm Arguments 114
Terminology 115
Algorithms ..... <algorithm> 115
Applying a Function to a Range 115
Checking for the Presence of Elements 117
Finding Elements 117
Finding Min/Max Elements 118
Binary Search 119
Subsequence Search 120
Sequence Comparison 121
Generating Sequences 122
Copy, Move, Swap 123
Removing and Replacing 124
Reversing and Rotating 125

Partitioning 126
Sorting 127
Sampling and Shuffling 128
Operations on Sorted Ranges 129
Permutation 130
Heaps 131
Numeric Algorithms ..... <numeric> 132
Reductions 132
Inner Products 133
Prefix Sums 134
Element Differences 135
Algorithms for Uninitialized Memory ..... <memory> 135
Parallel Algorithms C++17 ..... <execution> 136
Parallel Execution 137
Parallel Unsequenced Execution 138
Iterator Adaptors ..... <iterator> 138
Chapter 5: Input/Output 141
Input/Output with Streams 141
Helper Types ..... <ios> 142
Formatting Methods (std::ios_base) ..... <ios> 143
I/O Manipulators ..... <ios>, <iomanip> 145
Example 146
std::ios ..... <ios> 147
std::ostream ..... <ostream> 149
std::istream ..... <istream> 151
std::iostream ..... <istream> 153

String Streams ..... <sstream> 153
Example 154
File Streams ..... <fstream> 155
Example 156
Streaming Custom Types 156
Custom << and >> Operators 156
Custom I/O Manipulators ..... <ios> 157
Stream Iterators ..... <iterator> 160
std::ostream_iterator 160
std::istream_iterator 160
Stream Buffers ..... <streambuf> 161
File Systems ..... <filesystem> 162
Files, Paths, and Pathnames 162
Error Reporting 163
The path Class 164
File Links 168
Path Normalization 169
The Current Working Directory 170
Absolute and Relative Paths 170
Comparing Paths 172
File Status 172
Creating, Copying, Deleting, and Renaming 176
File Sizes and Free Space 177
Directory Listing 178
C-Style File Utilities ..... <cstdio> 180
C-Style Output and Input ..... <cstdio> 181
std::printf() Family 181
std::scanf() Family 185

Chapter 6: Characters and Strings 189

Strings ..... <string> 189

Searching in Strings 190

Modifying Strings 191

Constructing Strings 192

String Length 192

Copying (Sub)Strings 193

Comparing Strings 193

String Views C++17 ..... <string_view> 194

Character Classification ..... <cctype>, <cwctype> 195

Character-Encoding Conversion ..... <locale>, <codecvt> 197

Localization ..... <locale> 200

Locale Names 200

The Global Locale 201

Basic std::locale Members 202

Locale Facets 202

Combining and Customizing Locales 210

C Locales ..... <clocale> 213

Regular Expressions ..... <regex> 214

The ECMAScript Regular Expression Grammar 214

Regular Expression Objects 216

Matching and Searching Patterns 218

Match Iterators 221

Replacing Patterns 223

Numeric Conversions 226

Convenient Conversion Functions ..... <string> 227

High-Performance Conversion Functions C++17 ..... <charconv> 229

Chapter 7: Concurrency 231

Threads ..... <thread> 231

Launching a New Thread 231

A Thread's Lifetime 232

Thread Identifiers 232

Utility Functions 233

Exceptions 233

Futures ..... <future> 234

Return Objects 234

Providers 235

Exceptions 237

Mutual Exclusion ..... <mutex> 238

Mutexes and Locks 238

Mutex Types 239

Lock Types 241

Locking Multiple Mutexes 244

Exceptions 244

Calling a Function Once ..... <mutex> 245

Condition Variables ..... <condition_variable> 246

Waiting for a Condition 246

Notification 247

Exceptions 248

L1 Data Cache Line Size C++17 ..... <new> 248

Synchronization 249

Atomic Operations ..... <atomic> 250

Atomic Variables 250

Atomic Flags 255

Non-member Functions and Macros 255

Fences 255

Chapter 8: Diagnostics 257

Assertions ..... <cassert> 257

Exceptions ..... <exception>, <stdexcept> 258

Exception Pointers ..... <exception> 259

Nested Exceptions ..... <exception> 260

System Errors ..... <system_error> 262

std::error_category 263

std::error_code 263

std::error_condition 264

C Error Numbers ..... <cerrno> 264

Failure Handling ..... <exception> 265

std::uncaught_exceptions() C++17 265

std::terminate() 266

Appendix: Standard Library Headers 271

Numerics and Math (Chapter 1) 271

General Utilities (Chapter 2) 272

Containers (Chapter 3) 273

Algorithms (Chapter 4) 274

Input/Output (Chapter 5) 274

Characters and Strings (Chapter 6) 275

Concurrency (Chapter 7) 276

Diagnostics (Chapter 8) 277

The C Standard Library 277

Index 279

About the Authors

Peter Van Weert is a Belgian software engineer and C++ expert, mainly experienced in large-scale desktop application development. He is passionate about coding, algorithms, and data structures.

Peter received his master of science in computer science summa cum laude with congratulations of the Board of Examiners from the University of Leuven. In 2010, he completed his PhD thesis in Leuven at the research group for declarative languages and artificial intelligence. During his doctoral studies, he was a teaching assistant for courses on software analysis and design, object-oriented programming (Java), and declarative programming (Prolog and Haskell).

After graduating, Peter joined Nikon Metrology to work on industrial metrology software for high-precision 3D laser scanning and point cloud–based inspection. At Nikon, he learned to handle large C++ code bases and gained further proficiency in all aspects of the software development process—skills that serve him well today at Medicim, the software R&D center for dental companies Nobel Biocare, Ormco, and KaVo Kerr. At Medicim, Peter contributes to their next-generation digital platform for dentists, orthodontists, and oral surgeons that offers patient data acquisition from a wide range of hardware, diagnostic functionality, implant planning, and prosthetic design.

In his spare time, Peter writes books on C++ and is a regular speaker at and board member of the Belgian C++ Users Group.

Marc Gregoire is a software architect from Belgium. He graduated from the University of Leuven, Belgium, with a degree in "Burgerlijk ingenieur in de computer wetenschappen" (equivalent to a master of science in engineering in computer science). The year after, he received an advanced master's degree in artificial intelligence, cum laude, at the same university. After his studies, Marc started working for a software consultancy company called Ordina Belgium. As a consultant, he worked for Siemens and Nokia Siemens Networks on critical 2G and 3G software running on Solaris for telecom operators. This required working in international teams stretching from South America and the United States to Europe, the Middle East, Africa, and Asia. Now, Marc is a software architect at Nikon

Metrology (www.nikonmetrology.com), a division of Nikon and a leading provider of precision optical instruments and metrology solutions for 3D geometric inspection.

His main expertise is C/C++, specifically Microsoft VC++ and the MFC framework. He has experience in developing C++ programs running 24/7 on Windows and Linux platforms, for example, KNX/EIB home automation software. In addition to C/C++, Marc also likes C#.

Since April 2007, he has received the annual Microsoft MVP (Most Valuable Professional) award for his Visual C++ expertise.

Marc is the founder of the Belgian C++ Users Group (www.becpp.org), author of *Professional C++* (Wiley), technical editor for numerous books for several publishers, and a member on the CodeGuru forum (as Marc G). He maintains a blog at www.nuonsoft.com/blog/.

About the Technical Reviewer

Christophe Pichaud is a French C/C++ developer based in Paris. Over the course of his career, he has developed large scale server implementations in the banking industry, where he helped build the first French online bank account service (for Banque-Populaire), as well as Retail Services (Société Générale). He's also performed C++ migrations and developed hybrid applications with the .NET stack. Among his past clients are Accenture, Avanade, Sogeti, CapGemini, Palais de Elysée (French Presidency), SNCF, Total, Danone, CACIB, and BNP Paribas. He earned his MCSD.NET certification and currently works for a Microsoft Gold Partner called Devoteam Modern Applications in Paris, a division of Devoteam (www.devoteam.com).

Additionally, he participates in Microsoft Events as speaker for TechDays, and as an MVP at Ask the Expert sessions. He's regularly written C++ technical articles for the French magazine *Programmez* since 2011. He is also the community manager of the ".NET Azure Rangers," which includes 26 members and 9 MVPs and whose activities include speaking, writing and community-building around Microsoft technologies.

When he is not developing software or reading books, Christophe spends his spare time and holidays with his three daughters, Edith, Lisa, and Audrey along with his father Jean-Marc and mother Mireille in the Burgundy region of France.

Introduction

The C++ Standard Library

The C++ Standard Library is a collection of essential classes and functions used by millions of C++ programmers on a daily basis. Being part of the ISO Standard of the C++ Programming Language, an implementation is distributed with virtually every C++ compiler. Code written with the C++ Standard Library is therefore portable across compilers and target platforms.

The Library is more than 25 years old. Its initial versions were heavily inspired by a (then proprietary) C++ library called the *Standard Template Library (STL)*, so much so that many still incorrectly refer to the Standard Library as "the STL." The STL library pioneered generic programming with templated data structures called *containers* and *algorithms*, glued together with the concept of *iterators*. Most of this work was adapted by the C++ standardization committee, but nevertheless neither library is a true superset of the other.

Today the C++ Standard Library is much more than the containers and algorithms of the STL, though. For decades, it has featured STL-like string classes, extensive localization facilities, and a stream-based I/O library, as well as the entire C Standard Library. Earlier this decade, the C++11 and C++14 editions of the ISO standard have added, among other things, hash map containers, generic smart pointers, a versatile random number generation framework, a powerful regular expression library, more expressive utilities for function-style programming, type traits for template metaprogramming, and a portable concurrency library featuring threads, mutexes, condition variables, and atomic variables. Most recently, C++17 has introduced, among many smaller additions, parallelized algorithms, a file system library, and several key types for day-to-day use (such as `optional<>`, `variant<>`, `any`, and `string_view`). Many of the C++11, C++14, and C++17 additions are based on Boost, a collection of open source C++ libraries.

And this is just the beginning: the C++ community has rarely been as active and alive as in the past decade. The next version of the Standard, tentatively called C++20, is expected to add even more essential classes and functions.

Why This Book?

Needless to say, it is hard to know and remember all the possibilities, details, and intricacies of the vast and growing C++ Standard Library. This handy reference guide offers a condensed, well-structured summary of all essential aspects of the C++ Standard Library and is therefore indispensable to any C++ programmer.

You could consult the Standard itself, but it is written in a very detailed, technical style and is primarily targeted at Library implementors. Moreover, it is very long: the C++ Standard Library chapters alone are nearly 1,000 pages in length, and those on the C Standard Library easily encompass another 200 pages. Other reference guides exist but are often outdated, limited (most cover little more than the STL containers and algorithms), or not much shorter than the Standard itself.

This book covers all important aspects of the C++17 and C18 Standard Libraries, some in more detail than others, and is always driven by their practical usefulness. You will not find page-long, repetitive examples; obscure, rarely used features; or bloated, lengthy explanations that could be summarized in just a few bullets. Instead, this book strives to be exactly that: a summary. Everything you need to know and watch out for in practice is outlined in a compact, to-the-point style, interspersed with practical tips and short, well-chosen, clarifying examples.

Who Should Read This Book?

The book is targeted at all C++ programmers, regardless of their proficiency with the language or the Standard Library. If you are new to C++, its tutorial aspects will quickly bring you up to speed with the C++ Standard Library. Even the most experienced C++ programmer, however, will learn a thing or two from the book and find it an indispensable reference and memory aid. The book does not explain the C++ language or syntax itself, but is accessible to anyone with basic C++ knowledge or programming experience.

What You Will Learn

- How to use the powerful random number generation facilities
- How to work with dates and times
- What smart pointers are and how to use them to prevent memory leaks
- How to use containers to efficiently store and retrieve your data
- How to use algorithms to inspect and manipulate your data
- How lambda expressions allow for elegant use of algorithms
- What functionality the library provides for stream-based I/O
- How to inspect and manipulate files and directories on your file system
- How to work with characters and strings in C++
- How to write localized applications

- How to write safe and efficient multithreaded code using the C++11 concurrency library
- How to correctly handle error conditions and exceptions
- And more!

General Remarks

- All types, classes, functions, and constants of the C++ Standard Library are defined in the `std` namespace (short for *standard*).
- All C++ Standard Library headers must be included using `#include <`*header*`>` (note: no `.h` suffix!).
- All headers of the C Standard Library are available to C++ programmers in a slightly modified form by including `<c`*header*`>` (note the `c` prefix).[1] The most notable difference between the C++ headers and their original C counterparts is that all functionality is defined in the `std` namespace. Whether it is also provided in the global namespace is up to the implementation: portable code should therefore use the `std` namespace at all times.
- This book generally only covers headers of the C Standard Library if the C++ Standard Library does not offer more modern alternatives.
- More advanced, rarely used topics such as custom memory allocators are not covered.

Code Examples

To compile and execute the code examples given throughout the book, all you need is a sufficiently recent C++ compiler. We leave the choice of compiler entirely up to you, and we further assume you can compile and execute basic C++ programs. All examples contain standard, portable C++ code only and should compile with any C++ compiler that is compliant with the C++17 version of the Standard. We occasionally indicate known limitations of major compilers, but this is not a real goal of this book. In case of problems, please consult your implementation's documentation.

Nearly all code examples can be copied as is and put inside the `main()` function of a basic command-line application. Generally, only two headers have to be included to make a code snippet compile: the one being discussed in the context where the

[1] The original C headers may still be included with `<`*header*`.h>`, but their use is deprecated.

example is given and `<iostream>` for the command-line output statements (explained shortly). If any other header is required, we try to indicate so in the text. Should we forget, the Appendix provides a brief overview of all headers of the Standard Library and their contents. Additionally, you can download compilable source code files for all code snippets from this book from the Apress web site (www.apress.com/9781484218754).

Following is the obligatory first example (for once, we show the full program):

```
#include <iostream>

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

Many code samples, including those in earlier chapters, write to the standard output console using `std::cout` and the `<<` *stream insertion operator*, even though these facilities of the C++ I/O library are only discussed in detail in Chapter 5. The stream insertion operator can be used to output virtually all fundamental C++ types, and multiple values can be written on a single line. The so-called *I/O manipulator* `std::endl` outputs the newline character (`\n`) and flushes the output for `std::cout` to the standard console. Straightforward usage of the `std::string` class defined in `<string>` may occur in earlier examples as well. For instance:

```
std::string piString = "PI";
double piValue = 3.14159;
std::cout << piString << " = " << piValue << std::endl;  // PI = 3.14159
```

More details regarding streams and strings are found in Chapters 5 and 6, respectively, but this should suffice to get you through the examples in earlier chapters.

Common Class

The following Person class is used in code examples throughout the book to illustrate the use of user-defined classes together with the Standard Library:

```
#include <string>
#include <ostream>

class Person {
public:
   Person() = default;
   explicit Person(std::string first, std::string last = "",
                   bool isVIP = false)
   : m_first(move(first)), m_last(move(last)), m_isVIP(isVIP) {}

   const std::string& GetFirstName() const { return m_first; }
   void SetFirstName(std::string first) { m_first = move(first); }

   const std::string& GetLastName() const { return m_last; }
   void SetLastName(std::string last) { m_last = move(last); }

   bool IsVIP() const { return m_isVIP; }

private:
   std::string m_first, m_last;
   bool m_isVIP = false;
};

// Comparison operator
bool operator<(const Person& lhs, const Person& rhs) {
   if (lhs.IsVIP() != rhs.IsVIP()) return rhs.IsVIP();
   if (lhs.GetLastName() != rhs.GetLastName())
      return lhs.GetLastName() < rhs.GetLastName();
   return lhs.GetFirstName() < rhs.GetFirstName();
}

// Equality operator
bool operator==(const Person& lhs, const Person& rhs) {
   return lhs.IsVIP() == rhs.IsVIP()
       && lhs.GetFirstName() == rhs.GetFirstName()
       && lhs.GetLastName() == rhs.GetLastName();
}

// Stream insertion operator for output to C++ streams.
// Details of this operator can be found in Chapter 5.
std::ostream& operator<<(std::ostream& os, const Person& person) {
   return os << person.GetFirstName() << ' ' << person.GetLastName();
}
```

CHAPTER 1

Numerics and Math

Common Mathematical Functions <cmath>

The <cmath> header defines an extensive collection of common math functions in the std namespace. Unless otherwise specified, all functions are overloaded to accept all standard numerical types, with the following rules for determining the return type:

- If all arguments are float, the return type is float as well. Analogous for double and long double inputs.
- If mixed types or integers are passed, these numbers are converted to double, and a double is returned as well. If one of the inputs is a long double, long double is used instead.

Basic Functions

Function	Description
abs(x) fabs(x) fabsf(x) fabsl(x)	Returns the absolute value of x. abs() and fabs() accept all numeric types; fabsf() and fabsl() only float and long double. Starting with C++17, abs() no longer converts integers into doubles (as is conventional for <cmath>: see earlier). Instead, it behaves as abs() in <cstdlib> for integral x's (explained next). C++17
abs(x) labs(x) llabs(x)	Defined by <cstdlib>. Returns absolute value for an integral x. abs() accepts int, long, or long long (smaller integral types are promoted to int); labs() and llabs() only long and long long. The result has the same (possibly promoted) type as the input.
fmod(x, y) remainder(x, y)	Returns the remainder of x/y. For fmod(), the result always has the same sign as x; for remainder() that is not necessarily true. For example: mod(1,4) = rem(1,4) = 1, but mod(3,4) = 3 and rem(3,4) = -1.
remquo(x, y, *q)	Returns the same value as remainder(). q is a pointer to an int and receives a value with the sign of x/y and at least the last three bits of the integral quotient itself (rounded to nearest).

(*continued*)

P. Van Weert and M. Gregoire, *C++17 Standard Library Quick Reference*,
https://doi.org/10.1007/978-1-4842-4923-9_1

Function	Description

`div(x, y)`
`ldiv(x, y)`
`lldiv(x, y)`
	Defined by `<cstdlib>`. Returns a struct with two members, `quot` and `rem`, containing respectively `x / y` and `x % y` (though often computed with one instruction). `div()` accepts a pair of `int`s, `long`s, or `long long`s; `ldiv()` two `long`s, and `lldiv()` two `long long`s. The results have the same (possibly promoted) type as the inputs.

`fma(x, y, z)`	Computes $(x * y) + z$ in an accurate (better precision and rounding properties than a naïve implementation) and efficient (uses a single hardware instruction if possible) manner.
`fmin(x, y)` `fmax(x, y)`	Returns the minimum or maximum of `x` and `y`. `std::min()` and `max()` defined in `<algorithm>` are often more convenient, as they do not convert integers into `double`. These are explained later in this chapter.
`fdim(x, y)`	Returns the positive difference, i.e., $\begin{cases} x - y \text{ if } x > y \\ +0 \text{ if } x \le y \end{cases}$

`nan(string)`
`nanf(string)`
`nanl(string)`
	Returns a quiet (nonsignaling) NaN (Not-a-Number) of type `double`, `float`, `long double`, respectively, if available (`0` otherwise). The `string` parameter is an implementation-dependent tag that can be used to differentiate between different NaN values. Both `""` and `nullptr` are valid and result in a generic quiet NaN.

Exponential and Logarithmic Functions

Function	Formula	Function	Formula	Function	Formula
`exp(x)`	e^x	`exp2(x)`	2^x	`expm1(x)`	$e^x - 1$
`log(x)`	$\ln x = \log_e x$	`log10(x)`	$\log_{10} x$	`log2(x)`	$\log_2 x$
`log1p(x)`	$\ln(1 + x)$				

Power Functions

Function	Formula	Function	Formula
`pow(x, y)`	x^y	`sqrt(x)`	$\sqrt{x}$
`hypot(x, y)`	$\sqrt{x^2 + y^2}$	`cbrt(x)`	$\sqrt[3]{x}$
`hypot(x, y, z)`	$\sqrt{x^2 + y^2 + z^2}$ C++17		

Trigonometric and Hyperbolic Functions

<cmath> provides all basic trigonometric (sin(), cos(), tan(), asin(), acos(), atan()) and hyperbolic functions (sinh(), cosh(), tanh(), asinh(), acosh(), atanh()). All angles are expressed in radians.

The lesser-known trigonometric function atan2() is available as well. You use it to compute the angle between a vector (x, y) and the positive X axis. atan2(y, x) is similar to atan(y / x) except that its result correctly reflects the quadrant the vector is in (and that it also works if x is zero). Essentially, by dividing y by x in atan(y / x), one loses information regarding the sign of x and y.

Integral Rounding of Floating-Point Numbers

Function	Description
ceil(x) floor(x)	Rounds up/down to an integer. That is, returns the nearest integer that is not less/not greater than x.
trunc(x)	Returns the nearest integer not greater in absolute value than x.
round(x) lround(x) llround(x)	Returns the integral value nearest to x, rounding halfway cases away from zero. The return type of round() is based as usual on the type of x, while lround() returns long, and llround() returns long long.
nearbyint(x)	Returns the integral value nearest to x as a floating-point type. The current rounding mode is used: see round_style in the section on arithmetic type properties later in this chapter.
rint(x) lrint(x) llrint(x)	Returns the integral value nearest to x, using the current rounding mode. The return type of rint() is based as usual on the type of x, while lrint() returns long, and llrint() returns long long.

Floating-Point Manipulation Functions

Function	Description
modf(x, *p)	Breaks the value of x into an integral and fractional part. The latter is returned, the former is stored in p, both with the same sign as x. The return type is based on that of x as usual, and p must point to a value of the same type as this return type.
frexp(x, *exp)	Breaks the value of x into a normalized fraction with an absolute value in the range [0.5, 1) or equal to zero (the return value), and an integral power of 2 (stored in exp), with $x = fraction * 2^{exp}$.
logb(x)	Returns the floating-point exponent of x, i.e., $\log_{radix}\lvert x\rvert$, with *radix* the base used to represent floating-point values (2 for all standard numerical types, hence the name 'binary logarithm').
ilogb(x)	Same as logb(x) but the result is truncated to a signed int.

(*continued*)

Function	Description
`ldexp(x, n)`	Returns $x * 2^n$ (with `n` an `int`).
`scalbn(x, n)` `scalbln(x, n)`	Returns $x * radix^n$ (with `n` an `int` for `scalbn()` and a `long` for `scalbln()`). Radix is the base used to represent floating-point values (`2` for all standard C++ numerical types).
`nextafter(x, y)` `nexttoward(x, y)`	Returns the next representable value after `x` in the direction of `y`. Returns `y` if `x` equals `y`. For `nexttoward()`, the type of `y` is always `long double`.
`copysign(x, y)`	Returns a value with the absolute value of `x` and the sign of `y`.

Classification and Comparison Functions

Function	Description
`fpclassify(x)`	Classifies the floating-point value `x`: returns an `int` equal to `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, `FP_ZERO`, or an implementation-specific category.
`isfinite(x)`	Returns `true` if `x` is finite, i.e., normal, subnormal (denormalized), or zero, but not infinite or not-a-number.
`isinf(x)`	Returns `true` if `x` is positive or negative infinity.
`isnan(x)`	Returns `true` if `x` is not-a-number.
`isnormal(x)`	Returns `true` if `x` is normal, i.e., neither zero, subnormal (denormalized), infinite, nor not-a-number.
`signbit(x)`	Returns a nonzero value if `x` is negative.

`isgreater(x, y)`
`isgreaterequal(x, y)`
`isless(x, y)`
`islessequal(x, y)`
`islessgreater(x, y)`
	Compares `x` and `y`. The names are self-explanatory, except `islessgreater()` which returns `true` if $x < y \| \| x > y$. Note that this is not the same as `!=`, as, e.g., `nan("") != nan("")` is `true`, but not `islessgreater(nan(""), nan(""))`.

`isunordered(x, y)`	Returns whether `x` and `y` are unordered, i.e., whether one or both are not-a-number.

gcd/lcm C++17 <numeric>

The functions `gcd()` and `lcm()` compute the greatest common divisor and least common multiple, respectively. They are defined as follows:

```
template<typename M, typename N>
constexpr std::common_type_t<M, N> gcd(M, N);
```

```
template<typename M, typename N>
constexpr std::common_type_t<M, N> lcm(M, N);
```

Both `M` and `N` must be integer types. As explained in Chapter 2, `std::common_type_t<M, N>` is a so-called *type trait*, which in this case results in a type that both `M` and `N` can implicitly be converted to. Concretely, the common type of two integer types `M` and `N` is determined by the following rules (applied in order):

- If `N` and `M` are equal, their common type is that same type as well.
- If `N` and `M` are both smaller than `int`, their common type is `int`.
- If the size of `N` and `M` differs, their common type is the largest type.
- Otherwise, the common type is the one that is `unsigned`.

Error Handling

The mathematical functions from `<cmath>` can report errors in two ways depending on the value of `math_errhandling` (defined in `<cmath>`, although not in the `std` namespace). It has an integral type and can have one of the following values or their bitwise OR combination:

- `MATH_ERRNO`: Use the global `errno` variable (see Chapter 8).
- `MATH_ERREXCEPT`: Use the floating-point environment, `<cfenv>`, not further discussed in this book.

Special Mathematical Functions C++17 `<cmath>`

C++17 adds a collection of specialized mathematical functions. All of these are available in multiple overloads. In the following table, the functions without an asterisk always return a `double`. For the functions marked with an asterisk, the return type is always `double`, unless one of its arguments is a `long double`, then the return type is `long double` as well.

Additionally, there are two extra versions of each function with a postfix `f` or `l`. These additional functions accept `float`s and return a `float` (`f` postfix), or accept `long double`s and return a `long double` (`l` postfix). For example, `assoc_laguerre()`, `assoc_laguerref()`, and `assoc_laguerrel()`.

Explaining all the details of these mathematical functions falls outside the scope of this book. The following table just shows the mathematical formula for each function. Please consult a mathematical reference for more details.

■ **Note** At the time of writing, libc++, the implementation that ships with the Clang compiler, has not implemented these special mathematical functions yet.

Bessel Functions

Function	Description
`cyl_bessel_j(`v`, x)*`	Computesv thev cylindrical Bessel function of the first kind: $J_v(x)=\sum_{k=0}^{\infty}\frac{(-1)^k(x/2)^{v+2k}}{k!\Gamma(v+k+1)},\ \text{for}\ \lvert x\rvert\geq 0$
`cyl_neumann(`v`, x)*`	Computes the cylindrical Neumann function, also known as the cylindrical Bessel function of the second kind: $N_v(x)=\begin{cases}\frac{J_v(x)\cos v\pi - J_{-v}(x)}{\sin v\pi}, & \text{for } x\geq 0 \text{ and non-integral } v\\ \lim_{\mu\to v}\left(\frac{J_\mu(x)\cos\mu\pi - J_{-\mu}(x)}{\sin\mu\pi}\right), & \text{for } x\geq 0 \text{ and integral } v\end{cases}$
`cyl_bessel_i(`v`, x)*`	Computes the regular modified cylindrical Bessel function: $I_v(x)=i^{-v}J_v(ix)=\sum_{k=0}^{\infty}\frac{(x/2)^{v+2k}}{k!\Gamma(v+k+1)},\ \text{for}\ \lvert x\rvert\geq 0$
`cyl_bessel_k(`v`, x)*`	Computes the irregular modified cylindrical Bessel function: $K_v(x)=(\pi/2)i^{v+1}\left(J_v(ix)+iN_v(ix)\right)$ $=\begin{cases}\frac{\pi}{2}\frac{I_{-v}(x)-I_v(x)}{\sin v\pi}, & \text{for } x\geq 0 \text{ and non-integral } v\\ \frac{\pi}{2}\lim_{\mu\to v}\left(\frac{I_{-\mu}(x)-I_\mu(x)}{\sin\mu\pi}\right), & \text{for } x\geq 0 \text{ and integral } v\end{cases}$
`sph_bessel(n, x)`	Computes the spherical Bessel function of the first kind: $j_n(x)=\left(\frac{\pi}{2x}\right)^{1/2}J_{n+1/2}(x),\ \text{for}\ x\geq 0$
`sph_neumann(n, x)`	Computes the spherical Neumann function, also known as the spherical Bessel function of the second kind: $n_x(x)=\left(\frac{\pi}{2x}\right)^{1/2}N_{n+1/2}(x),\ \text{for}\ x\geq 0$

Polynomials

Function	Description
`legendre(l, x)`	Computes the Legendre polynomial of the first kind: $P_l(x)=\frac{1}{2^l l!}\frac{d^l}{dx^l}\left(x^2-1\right)^l,\ \text{for}\ \lvert x\rvert\le 1$
`assoc_legendre(l, m, x)`	Computes the associated Legendre function: $P_l^m(x)=\left(1-x^2\right)^{m/2}\frac{d^m}{dx^m}P_l(x),\ \text{for}\ \lvert x\rvert\le 1$
`sph_legendre(l, m, θ)`	Computes the spherical associated Legendre function $Y_l^m(\theta,0)$, where: $Y_l^m(\theta,\phi)=(-1)^m\left(\frac{(2l+1)}{4\pi}\frac{(l-m)!}{(l+m)!}\right)^{1/2}P_l^m(\cos\theta)e^{im\phi},$ for $\lvert m\rvert\le l$
`laguerre(n, x)`	Computes the Laguerre polynomial of order *n* at point *x*: $L_n(x)=\frac{e^x}{n!}\frac{d^n}{dx^n}\left(x^n e^{-x}\right),\ \text{for}\ x\ge 0$
`assoc_laguerre(n, m, x)`	Computes the associated Laguerre polynomial: $L_n^m(x)=(-1)^m\frac{d^m}{dx^m}L_{n+m}(x),\ \text{for}\ x\ge 0$
`hermite(n, x)`	Computes the Hermite polynomial of order *n* at point *x*: $H_n(x)=(-1)^n e^{x^2}\frac{d^n}{dx^n}e^{-x^2}$

Elliptic Integrals

Function	Description
`ellint_1(k, ϕ)*`	Computes the incomplete elliptic integral of the first kind: $F(k,\phi)=\int_0^\phi\frac{d\theta}{\sqrt{1-k^2\sin^2\theta}},\ \text{for}\ \lvert k\rvert\le 1$
`comp_ellint_1(k)`	Computes the complete elliptic integral of the first kind: $F\left(k,\frac{\pi}{2}\right),\ \text{for}\ \lvert k\rvert\le 1$
`ellint_2(k, ϕ)*`	Computes the incomplete elliptic integral of the second kind: $E(k,\phi)=\int_0^\phi\sqrt{1-k^2\sin^2\theta}\,d\theta,\ \text{for}\ \lvert k\rvert\le 1$

(*continued*)

Function	Description
`comp_ellint_2(k)`	Computes the complete elliptic integral of the second kind: $E\left(k,\frac{\pi}{2}\right),\ \text{for}\ \lvert k\rvert \le 1$
`ellint_3(k, ν, ϕ)*`	Computes the incomplete elliptic integral of the third kind: $\Pi(\nu,k,\phi)=\int_0^{\phi}\frac{d\theta}{\left(1-\nu\sin^2\theta\right)\sqrt{1-k^2\sin^2\theta}},\ \text{for}\ \lvert k\rvert \le 1$
`comp_ellint_3(k, ν)*`	Computes the complete elliptic integral of the third kind: $\Pi\left(k,\nu,\frac{\pi}{2}\right),\ \text{for}\ \lvert k\rvert \le 1$

Exponential Integrals

Function	Description
`expint(x)`	Computes the exponential integral: $Ei(x)=-\int_{-x}^{\infty}\frac{e^{-t}}{t}dt$

Error Functions

Function	Description
`erf(x)`	Computes the error function of a given value: $erf(x)=\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$
`erfc(x)`	Computes the complement of the error function of a given value: $erfc(x)=1-erf(x)=\frac{2}{\sqrt{\pi}}\int_x^{\infty}e^{-t^2}dt$

Gamma Functions

Function	Description
`tgamma(x)`	Computes the "true gamma" of a given value: $\Gamma(x)=\int_0^{\infty}t^{x-1}e^{-t}dt$
`lgamma(x)`	Computes: $\ln(\lvert\Gamma(x)\rvert)$

Beta Functions

Function	Description
`beta(x, y)*`	Computes the beta function: $B(x,y)=\frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$, for $x>0, y>0$

Zeta Functions

Function	Description
`riemann_zeta(x)`	Computes the Riemann zeta function: $\zeta(x)=\begin{cases}\sum_{k=1}^{\infty}k^{-x}, & \text{for } x>1\\ \frac{1}{1-2^{1-x}}\sum_{k=1}^{\infty}(-1)^{k-1}k^{-x}, & \text{for } 0\le x\le 1\\ 2^{x}\pi^{x-1}\sin\left(\frac{\pi x}{2}\right)\Gamma(1-x)\zeta(1-x), & \text{for } x<0\end{cases}$

Minimum, Maximum, and Clamping <algorithm>

The Standard Library provides a set of functions related to finding the minimum and maximum of two or more values. In the following function definitions, `T` is the element type, and the optional `Compare` parameter is a function object to be used to compare elements. `T` can be any type, not just a fundamental type. If no `Compare` function object is specified, `operator<` is used. The function object accepts two parameters and returns `true` if the first argument is less than the second, `false` otherwise. The ordering imposed must be a strict weak ordering, just as with the default `operator<`:

```
constexpr const T& min(const T& x, const T& y[, Compare comp])
constexpr const T& max(const T& x, const T& y[, Compare comp])
```

Returns a reference to the minimum or maximum of two values, or the first value if they are equal.

```
constexpr T min(initializer_list<T> list[, Compare comp])
constexpr T max(initializer_list<T> list[, Compare comp])
```

Returns a copy of the minimum or maximum value in a given `initializer_list` (discussed in Chapter 2), or a copy of the leftmost minimum or maximum if there are several elements equal to this extreme. Allows expressions of the form `std::min({ x, y, z })` to quickly determine the extreme of a limited number of elements.

```
constexpr pair<const T&, const T&>
   minmax(const T& x, const T& y[, Compare comp])
```

Returns a `pair` containing references to the minimum and maximum of two values, in that order. If `x` and `y` are equal, `pair(x, y)` is returned.

```
constexpr pair<T, T> minmax(initializer_list<T> list[, Compare comp])
```

Returns a `pair` containing a copy of the minimum and maximum values in an `initializer_list`, in that order. If several elements are equal to the minimum, then a copy of the leftmost one is returned; if several are equal to the maximum, a copy of the rightmost is returned.

C++17 also adds the following `std::clamp()` function which can be used to clamp, or bound, a given value to a given range C++17:

```
constexpr const T&
   clamp(const T& value, const T& low, const T& high[, Compare comp])
```

This function returns the following:

- A reference to `low` if `value` < `low`
- A reference to `value` if `low` ≤ `value` ≤ `high`
- A reference to `high` if `high` < `value`

Fixed-Width Integer Types <cstdint>

The `<cstdint>` header contains platform-dependent type aliases for integer types with different and more portable width requirements than the fundamental integer types:

- `std::(u)intX_t`, an (unsigned) integer of exactly *X* bits (*X* = 8, 16, 32, or 64). Present only if supported by the target platform.
- `std::(u)int_leastX_t`, the smallest (unsigned) integer type of at least *X* bits (*X* = 8, 16, 32, or 64).
- `std::(u)int_fastX_t`, the fastest (unsigned) integer type of at least *X* bits (*X* = 8, 16, 32, or 64).

- `std::(u)intmax_t`, the largest supported (unsigned) integer type.
- `std::(u)intptr_t`, (unsigned) integer type large enough to hold a pointer. These type aliases are optional.

The header further defines macros for the minimum and maximum values of these (and some other) types, for instance, `INT_FAST_8_MIN` and `INT_FAST_8_MAX` for `std::int_fast8_t`. The standard C++ way of obtaining these values though is with the facilities of `<limits>` discussed next.

Arithmetic Type Properties `<limits>`

The `std::numeric_limits<T>` template class offers a multitude of static functions and constants to obtain properties of a numeric type `T`. It is specialized for all fundamental numeric types, both integral and floating-point, and can hence be used to inspect properties of all their aliases as well, such as `size_t` or those of the previous section. The various members offered are listed as follows. Functions are only and always used to obtain a `T` value, whereas Booleans, `int`s, and `enum` values are defined as constants.

Member	Description
`is_specialized`	Indicates whether the template is specialized for the given type. If `false`, zero-initialized values are used for all other members.
`min()`, `max()`	Returns the minimum/maximum finite representable number. Rather unexpectedly, for floating-point numbers, `min()` returns the *smallest positive number* that can be represented (cf. `lowest()`).
`lowest()`	Returns the lowest finite representable number. Same as `min()` except for floating-point types, where `lowest()` returns the *lowest negative number*, which for `float` and `double` equals `-max()`.
`radix`	The base used to represent values (`2` for all C++ numerical types, but specific platforms could support, e.g., native decimal types).
`digits`	The number of digits in base `radix` (i.e., generally the number of bits) representable, *excluding any sign bit* for integer types. For floating-point types, the number of digits in the mantissa.
`digits10`	The number of significant decimal digits that the type can represent without loss, e.g., when converting *from* text and back. Equal to $\lfloor digits * \log_{10}(radix) \rfloor$ for integers: for `char`, e.g., it equals `2`, as it cannot represent all values with 3 decimal digits. For floating-point numbers, it equals $\lfloor (digits - 1) * \log_{10}(radix) \rfloor$.
`is_signed`	Identifies signed types. All standard floating-point types are signed, Booleans not, for `char` and `wchar_t` it is unspecified.
`is_integer`	Identifies integer types (includes Booleans and character types).

(*continued*)

Member	Description
`is_exact`	Identifies types with exact representations. Same as `is_integer` for all standard types, but there exist, e.g., third-party rational number representations that are exact but not integer.
`is_bounded`	Identifies types with finite representations. `true` for all standard types, but libraries exist that offer types with arbitrary precision.
`is_modulo`	Identifies 'modulo types', meaning that if the result of a `+`, `-`, or `*` operation would fall outside the range [`min()`, `max()`], the resulting value differs from the real value by an integral multiple of `max() - min() + 1`. Usually `true` for integers; `false` for floating-point types.
`traps`	Identifies types that have at least one value that would cause a trap (exception) when used as an operand for an arithmetic operation. For example, division by 0 always causes a trap. Usually `true` for all standard integer types, except `bool`. Usually `false` for all floating-point types.

The following members are relevant only for floating-point types. For integer types, they always equal or return zero:

Member	Description
`max_digits10`	The number of decimal digits needed to represent any value of the type without loss, e.g., when converting *to* text and back. Use (at least) `max_digits10` precision when converting floating-point numbers to text, and it will give the exact same value again when parsed back (`9` for `float`, `17` for `double`, `22` for `long double`).
`min_exponent10`, `min_exponent`, `max_exponent10`, `max_exponent`	The lowest negative (for `min_*`) or highest positive (for `max_*`) integer n such that 10^n (for `*10`) or $radix^{n-1}$ (otherwise) is a valid normalized floating-point value.
`epsilon()`	The difference between `1.0` and the next representable value.
`round_error()`	The maximum rounding error as defined in ISO/IEC 10967-1.
`is_iec599`	Identifies types conforming to all IEC 599/IEEE 754 requirements. Usually `true` for all standard floating-point types.
`has_infinity`	Identifies types that can represent positive infinity. Usually `true` for all standard floating-point types.
`infinity()`	Returns the value for positive infinity. Only meaningful if `has_infinity` is `true`.
`has_quiet_NaN`, `has_signaling_NaN`	Identifies types that can represent the special value for a quiet or signaling NaN (Not-a-Number). Usually `true` for all standard floating-point types. Using a signaling NaN in operations results in an exception; using a quiet NaN does not.

(*continued*)

Member	Description
`quiet_NaN()`, `signaling_NaN()`	Returns the value for a quiet or signaling NaN. Only meaningful if `has_quiet_NaN` respectively `has_signaling_NaN` is `true`.
`tinyness_before`	Identifies types that perform a check for underflow before performing any rounding.
`round_style`	Contains the rounding style as a `std::float_round_style` value: `round_indeterminate`, `round_toward_zero`, `round_to_nearest`, `round_toward_infinity`, or `round_toward_neg_infinity`. All integer types are required to round toward zero. The standard floating-point types usually round to nearest.
`has_denorm`	Identifies types that can represent denormalized values (special values smaller than `min()` that exist to deal with underflow). Has type `std::float_denorm_style`, with values `denorm_absent`, `denorm_present` (most common), and `denorm_indeterminate`.
`denorm_min()`	Returns smallest positive denormalized value if `has_denorm != std::denorm_absent`, and `min()` otherwise.
`has_denorm_loss`	Identifies types for which loss of precision is detected as denormalization loss rather than as an inexact result (advanced option which should be `false`; dropped in IEEE 754-2008).

Complex Numbers

<complex>

The `std::complex<T>` type, defined for at least `T` equal to `float`, `double`, and `long double`, is used to represent complex numbers as follows:

```
std::complex<float> c(1,2);   // Both arguments are optional (default: 0)
std::cout << "c=" << c.real() << '+'<< c.imag() << 'i' << '\n';   // c=1+2i
c.real(3); c.imag(3); c += 1;
std::cout << "norm(" << c << ") = " << std::norm(c);    // norm((4,3)) = 25
```

All expected operators are available: `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`, `=`, `==`, and `!=`, including overloads with a floating-point operand (which is then treated as a `complex` number with a zero imaginary part), and the `>>` and `<<` operators for interaction with the streams of Chapter 5.

The `std::literals::complex_literals` namespace defines convenient literal operators for creating `complex<T>` numbers: `i`, `if`, and `il`, creating values with `T` equal to `double`, `float`, and `long double`, respectively. Using this, the `c` value in the previous example could have been created with '`auto c = 1.f + 2if;`'.

The header furthermore defines the `complex` equivalents of several of the basic math functions seen earlier, that is, `pow()`, `sqrt()`, `exp()`, `log()`, and `log10()`, as well as all trigonometric and hyperbolic functions, that is, `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `sinh()`, `cosh()`, `tanh()`, `asinh()`, `acosh()`, and `atanh()`.

Besides these, the following `complex`-specific non-member functions exist:

Function	Description	Definition
`real()` / `imag()`	Non-member getters	*real* / *imag*
`abs()`	The *magnitude* or *modulus*	$\sqrt{real^2 + imag^2}$
`norm()`	The *norm*	$real^2 + imag^2$
`arg()`	The *phase* or *argument*	atan2(*imag*, *real*)
`conj()`	The *conjugate*	(*real*, –*imag*)
`polar()`	Construction from *polar coordinates* (m, ϕ) (= magnitude and phase)	$m(\cos\phi + i \sin\phi)$
`proj()`	Projection onto the Riemann sphere	(∞, ±0) if infinite *real* or *imag*; else (*real*, *imag*)

Compile-Time Rational Numbers `<ratio>`

The `std::ratio<Numerator,Denominator=1>` type template from the `<ratio>` header represents a rational number. What makes it peculiar is that it does so at the *type level*, rather than the usual *value level* (`std::complex` numbers are an example of the latter). While `ratio` values can be default-constructed, this is rarely the intention. Rather, the `ratio` type is generally used as type argument for other templates. For example, the `std::chrono::duration<T,Period=std::ratio<1>>` template explained in Chapter 2 may be instantiated as `duration<int,ratio<1,1000>>` to represent a duration of milliseconds, or as `duration<int,ratio<60>>` for a duration of minutes.

Convenience type aliases exist for all standard SI `ratio`s: `std::kilo`, for instance, is defined as `ratio<1000>`, and `std::centi` as `ratio<1,100>`. The full list is `atto` (10^{-18}), `femto` (10^{-15}), `pico` (10^{-12}), `nano` (10^{-9}), `micro` (10^{-6}), `milli` (10^{-3}), `centi` (10^{-2}), `deci` (10^{-1}), `deca` (10^{1}), `hecto` (10^{2}), `kilo` (10^{3}), `mega` (10^{6}), `giga` (10^{9}), `tera` (10^{12}), `peta` (10^{15}), and `exa` (10^{18}), and for platforms with an integer type that is wider than 64 bit `yocto` (10^{-24}), `zepto` (10^{-21}), `zetta` (10^{21}), and `yotta` (10^{24}).

All `ratio` types define two static members: `num` and `den`, containing the numerator and denominator of the rational number, but *after normalization*. The `ratio`'s `type` member equals the `ratio` type of this normalized rational number.

Arithmetic operations with `ratio`s are possible, but it is again at the type level: the `std::ratio_add` template, for instance, takes two `ratio` types as template arguments and evaluates to the type that corresponds to the sum of these rational numbers. The `ratio_subtract`, `ratio_multiply`, and `ratio_divide` templates are analogous. To compare two `ratio` types, similar `ratio_xxx` templates are provided with `xxx` either `equal`, `not_equal`, `less`, `less_equal`, `greater`, or `greater_equal`.

The following example clarifies `ratio` arithmetic (`<typeinfo>`, discussed in Chapter 2, must be included when using the `typeid` operator):

```
using a_third = std::ratio<1, 3>;
using a_half = std::ratio<1, 2>;
using two_quart = std::ratio<2, 4>;
using sum = std::ratio_add<a_third, a_half>;

std::cout << two_quart::num << '/' << two_quart::den << '\n';     // 1/2
std::cout << sum::num << '/' << sum::den << '\n';                 // 5/6
std::cout << std::boolalpha;        /* print true/false instead of 1/0 */
std::cout << (typeid(two_quart) == typeid(a_half)) << '\n';       // false
std::cout << (typeid(two_quart::type) == typeid(a_half)) << '\n'; // true
std::cout << std::ratio_equal<two_quart, a_half>::value << '\n';  // true
```

Random Numbers

`<random>`

The `<random>` library provides powerful random number generation facilities that supersede the flawed C-style `rand()` function from `<cstdlib>`. Central concepts are *random number generators* and *distributions*. A *generator* is a function object that generates random numbers in a predefined range in a uniformly distributed way—that is, each value in said range has, in principle, the same probability of being generated. A generator is generally passed to a *distribution* functor to generate random values distributed according to some chosen statistical distribution. This could, for instance, be another user-specified uniform distribution:

```
std::default_random_engine generator;
std::uniform_int_distribution<int> distribution(1, 6);
int dice_roll = distribution(generator);   // 1 <= dice_roll <= 6
```

When multiple values are to be generated, it is more convenient to bind the generator and distribution, for example, using the facilities of `<functional>` (Chapter 2):

```
std::function<int()> roller = std::bind(distribution, generator);
for (int i = 0; i < 100; ++i) std::cout << roller() << '\n';
```

Random Number Generators

The library defines two types of generators: *random number engines* that generate pseudorandom numbers and one *true nondeterministic random number generator*, `std::random_device`.

Pseudorandom Number Engines

Three families of pseudorandom number engines are provided in the form of generic class templates with various numeric type parameters:

- `std::linear_congruential_engine`: Uses a minimal amount of memory (one integer) and is reasonably fast, but generates low-quality random numbers.
- `std::mersenne_twister_engine`: Produces the highest-quality pseudorandom numbers, at the expense of a larger state size (the state of the predefined `mt19937` Mersenne Twister, e.g., consists of 625 integers). Still, as they are also the fastest generators, these engines should be your default choice if size is of no concern.
- `std::subtract_with_carry_engine`: While an improvement over the linear congruential engines in terms of quality (not speed though), these engines have much lower quality and performance than a Mersenne Twister. Their state size is more moderate though (96 bytes generally).

All these engines provide a constructor that accepts an optional seed to initialize the engine. Seeding is explained later. They also have a copy constructor and support the following operations:

Operation	Description
`seed(value)`	Reinitializes the engine by seeding it with a given `value`.
`operator()`	Generates and returns the next pseudorandom number.
`discard(n)`	Generates `n` pseudorandom numbers and discards them.
`min()` `max()`	Returns the minimum and maximum value that the engine can possibly generate.
`==` / `!=`	Compares the internal state of two engines (non-member operators).
`<<` / `>>`	Serialization to/from streams: see Chapter 5 (non-member operators).

All three engine templates require a series of numerical template parameters. Because choosing the appropriate parameters is best left to experts, several predefined instantiations exist for each family. Before we discuss these though, we first need to introduce *random number engine adaptors*.

Engine Adaptors

The following function objects *adapt* the output of an underlying engine:

- `std::discard_block_engine<e,p,r>`: For each block of `p > 0` generated numbers by the underlying engine `e`, it discards all but `r` kept values (with `p >= r > 0`).

- `std::independent_bits_engine<e,w>`: Generates random numbers of `w > 0` bits even if the underlying engine `e` produces numbers with a different width.
- `std::shuffle_order_engine<e,k>`: Delivers the numbers of the underlying engine `e` in a different, randomized order. Keeps a table of `k > 0` numbers, each time returning and replacing a random one of those.

All the adaptors have a similar set of constructors: a default constructor, one with a seed that is forwarded to the wrapped engine, and constructors that accept an lvalue or rvalue reference to an existing engine to copy or move.

Adaptors support the exact same operations as the wrapped engines, plus:

Operation	Description
`seed()`	Reinitializes the underlying engine by seeding it with a default seed.
`base()`	Returns a `const` reference to the underlying engine.

Predefined Engines

Based on the preceding engines and adaptors, the library provides the following predefined engines that you should use instead of using the engines and/or adaptors directly. The mathematical parameters for these have been defined by experts.

- `minstd_rand0` / `minstd_rand` are `linear_congruential_engines` that generate `std::uint_fast32_t` numbers in $[0, 2^{31}-1)$.
- `knuth_b` equals `shuffle_order_engine<minstd_rand0,256>`.
- `mt19937` / `mt19937_64` are `mersenne_twister_engines` generating `uint_fast32_t` / `uint_fast64_t` numbers.
- `ranlux24_base` / `ranlux48_base` are rarely used stand-alone (cf. next bullet), but are `subtract_with_carry_engines` that generate `uint_fast32_t` / `uint_fast64_t` numbers.
- `ranlux24` / `ranlux48` are `ranlux24_base` / `ranlux48_base` engines adapted by a `discard_block_engine`.

■ **Tip** Since choosing between all preceding predefined engines can still be daunting, an implementation must also offer a `std::default_random_engine` which should be good enough for most applications (it may be a type alias for one of the other engines).

Nondeterministic Random Number Generator

A `random_device`, in principle, does not generate pseudorandom numbers, but truly *nondeterministic* uniformly distributed random numbers. How it accomplishes this is implementation dependent: it could, for example, use special hardware on your CPU to generate numbers based on some physical phenomenon. If the `random_device` implementation cannot generate true nondeterministic random numbers, it is allowed to fall back to one of the pseudorandom number engines discussed earlier. To detect this, use its `entropy()` method: it returns a measure of the quality of the generated numbers, but zero if a pseudorandom number engine is used.

`random_device` is noncopyable and has but one constructor that accepts an optional implementation-specific `string` to initialize it. It has member functions `operator()`, `min()`, and `max()` analogous to the ones provided by the engines. Unlike for pseudorandom number engines before though, its `operator()` may throw an `std::exception` if it failed to generate a number (e.g., due to hardware failure).

While a `random_device` generates true random numbers, possibly cryptographically secure (check your library documentation), it is typically slower than any pseudorandom engine. It is therefore common practice to seed a pseudorandom engine using a `random_device`, as explained in the next section.

Seeding

All pseudorandom number engines have to be seeded with an initial value. If you set up an engine with the same seed, then you always get the same sequence of generated numbers. While this could be useful for debugging or for certain simulations, most of the time you want a different unpredictable sequence of numbers to be generated on each run. That is why it is important to seed your engine with a different value each time the program is executed. This has to be done once (e.g., at construction time). The recommended way of doing this is with a `random_device`, but as we saw earlier, this may also just generate pseudorandom numbers. A popular alternative is by seeding with the current time (cf. Chapter 2). For example:

```
std::random_device seeder;
const auto seed = seeder.entropy() ? seeder() : std::time(nullptr);
std::default_random_engine generator(
   static_cast<std::default_random_engine::result_type>(seed));
```

Random Number Distributions

Up to now, we have only talked about generating random numbers that are uniformly distributed in the full range of 32- or 64-bit unsigned integers. The library provides a large collection of distributions you can use to fit this distribution, range, and/or value type to your needs. Their names will sound familiar if you are fluent in statistics. While describing all maths behind them falls outside the scope of this book, the following sections briefly describe the available distributions, some in more detail than others. For each distribution, we show the supported constructors. For details on these distributions and their parameters, we recommend you consult a mathematical reference.

Uniform Distributions

```
uniform_int_distribution<Int=int>(Int a=0, Int b=numeric_limits<Int>::max())
uniform_real_distribution<Real = double>(Real a = 0.0, Real b = 1.0)
```

Generates uniformly distributed integer/floating-point numbers in the range [a, b] (both inclusive).

```
Real generate_canonical<Real, size_t bits, Generator>(Generator&)
```

This is the only distribution that is defined as a function instead of a functor. It generates numbers in the range [0.0, 1.0) using the given `Generator` as the source of the randomness. The `bits` parameter determines the number of bits of randomness in the mantissa.

Bernoulli Distributions

```
bernoulli_distribution(double p = 0.5)
```

Generates random Boolean values with `p` equal to the probability of generating `true`.

```
binomial_distribution<Int = int>(Int t = 1, double p = 0.5)
negative_binomial_distribution<Int = int>(Int k = 1, double p = 0.5)
geometric_distribution<Int = int>(double p = 0.5)
```

Generate random non-negative integral values according to a certain probability density function.

Normal Distributions

```
normal_distribution<Real = double>(Real mean = 0.0, Real stddev = 1.0)
```

Generates random numbers according to a normal, also called Gaussian, distribution. The parameters specify the expected `mean` and standard deviation `stddev`. In Figure 1-1 μ represents the mean and σ the standard deviation.

```
lognormal_distribution<Real = double>(Real mean = 0.0, Real stddev = 1.0)
chi_squared_distribution<Real = double>(Real degrees_of_freedom = 1.0)
cauchy_distribution<Real = double>(Real peak_location = 0., Real scale = 1.)
fisher_f_distribution<Real = double>(Real dof_num = 1., Real dof_denom = 1.)
student_t_distribution<Real = double>(Real degrees_of_freedom = 1.0)
```

Some more advanced normal-like distributions.

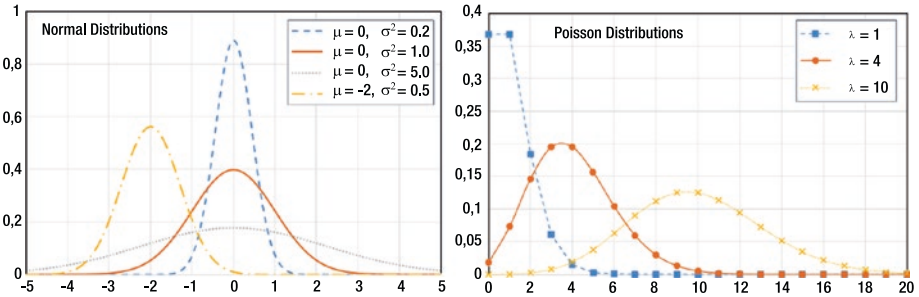


Figure 1-1. *Probability distributions for some example normal and Poisson distributions, plotting the probability (between 0 and 1) that a value is generated*

Poisson Distributions

```
poisson_distribution<Int = int>(double mean = 1.0)
exponential_distribution<Real = double>(Real lambda = 1.0)
gamma_distribution<Real = double>(Real alpha = 1.0, Real beta = 1.0)
weibull_distribution<Real = double>(Real a = 1.0, Real b = 1.0)
extreme_value_distribution<Real = double>(Real a = 0.0, Real b = 1.0)
```

Various distributions related to the classical Poisson distribution. The latter is illustrated in Figure 1-1, where λ is the `mean` (which for this distribution is equal to the variance). The Poisson distribution generates integers, so the connecting lines are there for illustration purposes only.

Sampling Distributions

Discrete Distribution

A discrete distribution requires a set of `count` weights and generates random numbers in the range [0, `count`). The probability of a value depends on its weight. The following constructors are provided:

```
discrete_distribution<Int = int>()
discrete_distribution<Int = int>(InputIt first, InputIt last)
discrete_distribution<Int = int>(initializer_list<double> weights)
discrete_distribution<Int = int>(size_t count, double xmin, double xmax,
                                 UnaryOperation op)
```

The default constructor initializes the distribution with a single weight of 1.0. The second and third constructors initialize it with a set of weights given as an iterator range, discussed in Chapter 3, or as an `initializer_list`, discussed in Chapter 2. And the last one initializes it with `count` weights generated by calling the given unary operation. The following formula is used:

$$weight_i = op\left(xmin + i * \delta + \frac{\delta}{2}\right) \text{ with } \delta = \frac{xmax - xmin}{count}$$

Piecewise Constant Distribution

A piecewise constant distribution requires a set of intervals and a weight for each interval. It generates random numbers which are uniformly distributed in each of the intervals. The following constructors are provided:

```
piecewise_constant_distribution<Real = double>()
```

The default constructor initializes the distribution with a single interval with boundaries 0.0 and 1.0, and weight 1.0.

```
piecewise_constant_distribution<Real = double>(
   InputIt1 firstBound, InputIt1 lastBound, InputIt2 firstWeight)
```

Initializes the distribution with intervals whose bounds are taken from the `firstBound`, `lastBound` iterator range and weights taken from the range starting at `firstWeight`.

```
piecewise_constant_distribution<Real = double>(
    initializer_list<Real> bounds, UnaryOperation weightOperation)
```

Initializes the distribution with intervals whose bounds are given as an `initializer_list` and weights generated by the given unary operation.

```
piecewise_constant_distribution<Real = double>(size_t count,
    Real xmin, Real xmax, UnaryOperation weightOperation)
```

Initializes the distribution with `count` uniform intervals over the range [`xmin`, `xmax`] and weights generated by the given unary operation.

The `piecewise_constant_distribution` has methods `intervals()` and `densities()` returning the interval boundaries and the probability densities for the values in each interval.

Piecewise Linear Distribution

A piecewise linear distribution, as implemented by `piecewise_linear_distribution`, is similar to a piecewise constant one, but has a linear probability distribution in each interval instead of a uniform one. It requires a set of intervals and a set of weights for each interval boundary. It also provides `intervals()` and `densities()` methods. The set of constructors is analogous to those discussed in the previous section, but one extra weight is required because each boundary needs a weight instead of each interval.

Example

```
std::mt19937 generator;           // Default-seeded for this example
std::vector intervals = { 1,20,40,60,80 };
std::vector weights = { 1,3,1,3} ;
std::piecewise_constant_distribution<double> distribution(
    begin(intervals), end(intervals), begin(weights));
int value = static_cast<int>(distribution(generator));
```

The graph on the left in Figure 1-2 shows the number of times a specific value has been generated when generating a million values using the preceding code. In the graph you clearly see the `piecewise_constant_distribution` with intervals (1,20), (20,40), (40,60), and (60,80) with interval weights 1, 3, 1, and 3.

The graph on the right shows a `piecewise_linear_distribution` with the same intervals and with boundary weights 1, 3, 1, 3, and 1. Notice that you require one extra weight compared to the `piecewise_constant_distribution` because you specify the weights for the boundaries instead of for the intervals.

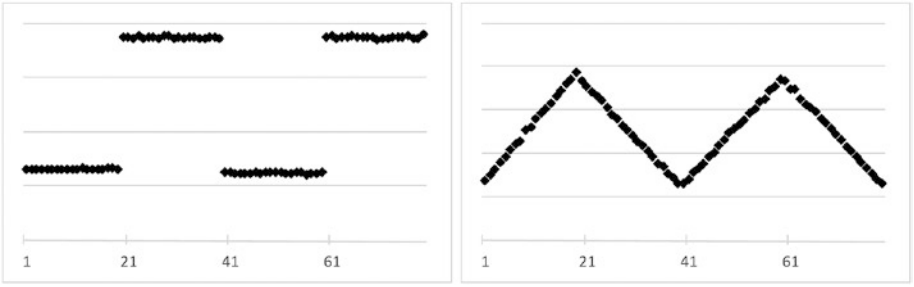


Figure 1-2. *Difference between a piecewise constant and piecewise linear distribution*

If you use a `piecewise_linear_distribution` with intervals of different sizes, the graph will not be continuous. That is because the weights are given for the boundaries of an interval, so if the beginning has a weight of 3 and the end has a weight of 1, then it means that the value at the beginning of the interval is three times more likely to be generated compared to the value at the end. Therefore, if the interval is, for example, twice as long, all probabilities will be twice as small as well, including those of the bounds.

Numeric Arrays <valarray>

std::valarray is a container-like class for storing and efficiently manipulating dynamic arrays of numeric values. A valarray has built-in support for multidimensional arrays and for efficiently applying most mathematical operations defined in <cmath> to each element. Types stored in a valarray must essentially be an arithmetic or pointer type, or a class that behaves similarly such as std::complex. Thanks to these restrictions, some compilers are able to optimize valarray calculations more than when working with other containers.

std::valarray provides the following constructors:

Constructor	Description
valarray() valarray(count)	Constructs an empty valarray, or one with count zero-initialized/default-constructed elements.
valarray(const T& val, n) valarray(const T* vals, n)	Constructs a valarray with n copies of val, or n copies from the vals array.
valarray(initializer_list)	Constructs a valarray and initializes it with the values from the initializer list.
valarray(const *x*_array<T>&)	Constructors that convert between *x*_array<T> and valarray<T>, where *x* can be slice, gslice, mask, or indirect. All four types are discussed later.
valarray(const valarray&) valarray(valarray&&)	Copy and move constructor.

Here is an example:

```
std::valarray<int> ints1(7);               // 7 zero-initialized integers
std::valarray doubles = { 1.1, 2.2, 3.3 }; // Deduces std::valarray<double>
int carray[] = { 6,5,4,3,2,1 };
std::valarray ints2(carray, 3);            // Contains 6,5,4
```

A valarray supports the following operations:

Operation	Description
operator[]	Retrieves a single element, or a part, i.e., a slice_array, gslice_array, mask_array or indirect_array discussed later.
operator=	Copy, move, and initializer-list assignment operators. You can also assign an instance of the element type: all elements in the valarray will be replaced with a copy of it.
operator+, -, ~, !	Applies unary operations to each element. Returns a new valarray with the result (operator! returns valarray<bool>).
operator+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Applies these operations to each element. Input is either 'const T&' or an equally long 'const valarray<T>&'. In the latter case, the operator is piecewise applied.

(*continued*)

Operation	Description
`swap()`	Swaps two `valarrays`.
`size()` `resize(n,val=T())`	Returns or changes the number of elements. When resizing you can specify the value to assign to new elements, they are zero-initialized by default.
`sum()`, `min()`, `max()`	Returns the sum, minimum, and maximum of all elements.
`shift(int n)` `cshift(int n)`	Returns a new `valarray` of the same size in which elements are shifted by `n` positions. If `n < 0`, elements are shifted to the left. Elements shifted out will be zero-initialized for `shift()`, while `cshift()` performs a circular shift.
`apply(func)`	Returns a new `valarray` where each element is calculated by applying the given unary function to the current elements.

The following non-member functions are supported as well:

Operation	Description
`swap()`	Swaps two `valarrays`.
`begin()`, `end()`	Returns begin and end iterators (cf. Chapters 3 and 4).
`abs()`	Returns a `valarray` with the absolute values.
`operator+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `&&`, `||`	Applies these binary operators to a `valarray` and a value, or to each element of two equally long `valarrays`.
`operator==`, `!=`, `<`, `<=`, `>`, `>=`	Returns a `valarray<bool>` where each element is the result of comparing elements of two `valarrays` or the elements of one `valarray` with a value.

There is also support for applying exponential (`exp()`, `log()`, and `log10()`), power (`pow()` and `sqrt()`), trigonometric (`sin()`, `cos()`, etc.), and hyperbolic (`sinh()`, `cosh()`, and `tanh()`) functions to all elements at once. These non-member functions return a new `valarray` with the results.

std::slice

This represents a *slice* of a `valarray`. A `std::slice` itself does not contain or refer to any elements, it simply defines a sequence of indices. These indices are not necessarily contiguous. It has three constructors: `slice(start, size, stride)`, a default constructor equivalent to `slice(0,0,0)`, and a copy constructor. Three getters are provided: `start()`, `size()`, and `stride()`. To use `slice`, create one and pass it to `operator[]` of a `valarray`. This selects `size()` elements from the `valarray` starting at position `start()`, with a given `stride()` (step size). If called on a `const valarray`, the result is a `valarray` with copies of the elements. Otherwise, it is a `slice_array` with references to the elements.

slice_array supports less operations than a valarray but can be converted to a valarray using the valarray(const slice_array<T>&) constructor. slice_array has the following three assignment operators:

```
void operator=(const T& value) const
void operator=(const valarray<T>& arr) const
const slice_array& operator=(const slice_array& arr) const
```

Operators +=, -=, *=, /=, %=, &=, |=, ^=, <<=, and >>= are provided as well. These operators require a right-hand side operand of the same type as the valarray to which the slice_array refers to, and apply the operator to the elements referred to by the slice_array. For example:

```
std::valarray ints = { 0,1,2,3,4,5,6,7 };
std::slice mySlicer(2, 3, 2);
const std::valarray<int>& constInts = ints;
auto copies = constInts[mySlicer];     // valarray<int> with copies of 2,4,6
auto refs = ints[mySlicer];     // slice_array<int> with references to 2,4,6
std::valarray factors{ 6,3,2 };
refs *= factors;                       // ints will be 0,1,12,3,12,5,12,7
```

One use case for slices is to select rows or columns from valarrays that represent matrices. They can also be used to implement matrix algorithms such as matrix multiplication.

std::gslice

gslice stands for *generalized slice*. Instead of having a single value for the size and stride, a gslice has a valarray<size_t> for sizes and one for strides. The default constructor is equivalent to gslice(0, valarray<size_t>(), valarray<size_t>()), and a copy constructor is provided as well. Just as std::slice, getters start(), size(), and stride() are available. Analogous to slice, a gslice is used by passing it to operator[] of valarray, returning either a valarray with copies or a gslice_array with references. A gslice_array supports a similar set of operations as a slice_array. How the different sizes and strides are used is best explained with an example:

```
std::valarray a = { 0,11,22,33,44,55,66,77,88,99,111 };
std::valarray<size_t> sizes{ 2,3 }; // <size_t> required here, otherwise it
                                    // deduces to std::valarray<int>.
std::valarray<size_t> strides{ 5,2 };
std::valarray r = a[std::gslice(1,sizes,strides)]; //11,33,55,66,88,111
```

In this example we have two values for size and stride, so the `gslice` will create two slices. The first slice has the following parameters:

- Start index = 1 (the first argument to the `gslice` constructor)
- Size = 2 and stride = 5 (the first values in `sizes` and `strides`)

This slice therefore represents the indices {1, 6}. With this, two second-level slices are created, one for each of these indices. The indices from the first-level slice are used as starting indices for the two second-level slices. The first second-level slice therefore has parameters:

- Start index = 1 (the first index of the first slice {1, 6})
- Size = 3 and stride = 2 (second values from `sizes` and `strides`)

and the second (note that both have the same size and stride parameters):

- Start index = 6 (the second index of the first slice {1, 6})
- Size = 3 and stride = 2 (second values from `sizes` and `strides`)

Concatenated, the second-level slices therefore represent these indices: {1,3,5, 6,8,10}. If there were a third level (i.e., third values in `sizes` and `strides`), these indices would serve as starting indices for six third-level slices (all using those third values of `sizes` and `strides`). Because there is no third level in this example, the corresponding values are simply selected from the `valarray`: {11,33,55, 66,88,111}.

std::mask_array

The `operator[]` on a `valarray` also accepts a `valarray<bool>`, similarly returning either a `valarray` with copies or a `std::mask_array` with references. This operator selects all elements from a `valarray` that have a `true` value in the corresponding position in the `valarray<bool>`. A `mask_array` supports an analogous set of operations as a `slice_array`. Here is an example:

```
std::valarray ints = { 0,1,2,3,4,5,6,7,8,9,10 };
// Construct a valarray<bool> with true for all even elements in ints.
std::valarray even = ((ints % 2) == 0);
// Count the number of true values in even. (See Chapter 4)
auto count = std::count(begin(even), end(even), true);
// Construct a valarray<int> with count elements of value 4.
std::valarray factors(4, count);
// Multiply the even elements in ints with a factor of 4.
ints[even] *= factors;    // 0,1,8,3,16,5,24,7,32,9,40
```

std::indirect_array

Lastly, the `operator[]` on `valarray` accepts a `valarray<size_t>` as well, returning either a `valarray` with copies or a `std::indirect_array` with references. The `valarray<size_t>` specifies which indices should be selected. An `indirect_array` again supports an analogous set of operations as a `slice_array`. An example:

```
std::valarray ints = { 0,1,2,3,4 };
std::valarray<size_t> indices = { 1,3,4 }; // <size_t> required here.
ints[indices] = -1;                        // 0,-1,2,-1,-1
```

CHAPTER 2

General Utilities

Moving, Forwarding, Swapping <utility>

This section explains `move()`, `move_if_noexcept()`, `forward()`, `swap()`, and `exchange()`. In passing, we also introduce the concepts of *move semantics* and *perfect forwarding*.

Moving

An object can be *moved* elsewhere (rather than copied) if its previous user no longer needs it afterward. Moving the resources from one object to another can often be implemented far more efficiently than (deep) copying them. For a `string` object, for instance, moving is typically as simple as copying a `char*` pointer and a length (constant time); there is no need to copy the entire `char` array (linear time).

Unless otherwise specified, the source object that was moved from is left in an undefined but valid state and should not be used anymore unless reinitialized. A valid implementation for moving a `std::string` (cf. Chapter 6), for instance, could set the source's `char*` pointer to `nullptr` to prevent the array from being deleted twice, but this is not required by the standard. Likewise, it is unspecified what `length()` will return after being moved from. Some operations, such as assignments, remain allowed though as demonstrated in the following example:

```
void f(std::string s)   { std::cout << "Moved or copied: " << s << '\n'; }
void g(std::string&& s) { std::cout << "Moved " << s << '\n'; }
std::string h() { std::string s("test"); return s; } // moved implicitly,
int main() {                                          // =std::move(s)
  std::string test("123");
  f(test);               // test copied to a new string
  f(std::move(test));    // test moved to a new string (move constructor)
  // std::cout << test; --> Undefined: may give "", "123", or simply crash
  test = "456";          // test is reinitialized, and may be used again
  // g(test);           --> Does not compile
  g(std::move(test));
  g(std::string("789")); // Unnamed objects are moved implicitly
  g(h());
}
```


P. Van Weert and M. Gregoire, *C++17 Standard Library Quick Reference*,
https://doi.org/10.1007/978-1-4842-4923-9_2

■ **Tip** In Standard Library speak, unless otherwise specified, only operations *without preconditions* are guaranteed to work on an object after moving its contents. The reason is that *its class invariants* may no longer hold. If you do want to reuse such an object, look for a member named `clear()` or `reset()`: these normally have no preconditions and can be used to bring the object back to a well-defined initial state, where all class invariants are guaranteed to hold. These considerations hold for nearly all Standard Library types. The most notable exceptions are the smart pointers we explain later in this chapter: smart pointers are always reset to their `nullptr` state after moving.

Despite its name, the `std::move()` function technically does not move anything: instead, it simply marks that a given `T`, `T&`, or `T&&` value *may* be moved, by statically casting it to an *rvalue reference* `T&&`. Because of the type cast, other functions may get selected by overload resolution, and/or value parameter objects may become initialized using their *move constructors* (of form `T(T&& t)`), if available, rather than their copy constructors. This initialization occurs at the callee side, not the caller side. An *rvalue parameter* `T&&` forces the caller to always move.

Similarly, an object can also be moved to another using a *move assignment operator* (of form `operator=(T&&)`):

```
std::string one("Test 123");
std::string other;
other = std::move(one);
// std::cout << one; --> Undefinedbehavior: one was moved to other
```

If no move member is defined, either explicitly or implicitly, overload resolution for `T&&` will fall back to `T&` or `T`, and in the latter case still create a copy. Conditions for implicit move members to be generated include that there may not be any user-defined copy, move, or destructor members, nor any non-static member variable or base class that cannot be moved.

The `move_if_noexcept()` function is similar to `move()`, except that it only casts to `T&&` if the move constructor of `T` is known not to throw from its exception specification (`noexcept`, or the deprecated `throw()`); otherwise, it casts to `const T&`.

All classes defined by the standard have move members if appropriate. Many containers of Chapter 3, for example, can be moved in constant time (not `std::array`, although it will move individual elements if possible to avoid deep copies).

■ **Tip** If you define a move constructor or move assignment operator, always do so *with a* `noexcept` *specifier*. The container classes of Chapter 3 extensively use moving to speed up operations such as adding a new element or when relocating arrays of elements (e.g., with sequential containers). Similarly, many algorithms of Chapter 4 will benefit if efficient move members are available (and/or non-member `swap()` operations: cf. later). However, and

especially when moving arrays of elements, these optimizations often only take effect if the values' move members are known not to throw.

Forwarding

The `std::forward()` helper function is intended to be used in templated functions to efficiently pass its arguments along to other functions while preserving any move semantics. If the argument to `forward<T>()` was an lvalue reference `T&`, this reference is returned unchanged. Otherwise, the argument is cast to an rvalue reference `T&&`. An example will clarify its intended use:

```
struct A { A() {}; A(const A&) = delete; }; // A objects cannot be copied
void f(const A&) { std::cout << "lval, "; } // forwarded as lvalue ref
void f(A&&)      { std::cout << "rval, "; } // forwarded as rvalue ref

// Three different forwarding (fwd) schemes:
template <typename T> void good_fwd(T&& t) { f(std::forward<T>(t)); }
template <typename T> void bad_fwd(T&& t)  { f(t); }
template <typename T> void ugly_fwd(T t)   { f(t); }

int main() {
  A a;
  good_fwd(a); good_fwd(std::move(a)); good_fwd(A()); // lval, rval, rval,
  bad_fwd(a);  bad_fwd(std::move(a)); bad_fwd(A());   // lval, lval,lval,
// ugly_fwd(a); ugly_fwd(std::move(a));ugly_fwd(A()); --> error: 3x copy
}
```

The idiom used by `good_fwd()` is called *perfect forwarding*. It optimally preserves rvalue references (such as those of `std::move()`d or temporary objects). The idiom's first ingredient is a so-called *forwarding* or *universal reference*: a `T&&` parameter, with `T` a template type parameter. Without it, template argument deduction removes all references: for `ugly_fwd()`, both `A&` and `A&&` become `A`. With a forwarding reference, `A&` and `A&&` are deduced respectively: that is, even though the forwarding reference looks like `T&&`, if passed `A&`, `A&` is deduced and not `A&&`. Still, using a forwarding reference alone is not enough, as shown with `bad_fwd()`. When using the *named* variable `t` as is, it binds with an lvalue function parameter (all named variables will), even if its type is deduced as `A&&`. This is where `std::forward<T>()` comes in. Similar to `std::move()`, it casts to `T&&`, but only if given a value with an rvalue type (including named variables of type `A&&`).

All this is quite subtle and is more about the C++ language (type deduction in particular) than the Standard Library. The main takeaway here is that to correctly forward arguments of a function template to a function, you should consider using perfect forwarding—that is, a forwarding reference combined with `std::forward()`.

Swapping and Exchanging

The `std::swap()` template function swaps two objects as if implemented as

```
template<typename T> void swap(T& one, T& other)
{ T temp(std::move(one)); one = std::move(other); other = std::move(temp); }
```

A similar `swap()` function template to piecewise swap all elements of equally long `T[N]` arrays is defined as well.

While already quite efficient if proper move members are available, for truly optimal performance one should consider specializing these template functions, for instance, to eliminate the need to move to a temporary. Many algorithms of Chapter 4, for instance, will call this non-member `swap()` function. For standard types, `swap()` specializations are already defined where appropriate.

A function similar to `swap()` is `std::exchange()`, which assigns a new value to something while returning its old value. A valid implementation is

```
template<typename T, typename U=T> T exchange(T& x, U&& new_val)
  { T old_val(std::move(x)); x = std::forward<U>(new_val); return old_val; }
```

■ **Tip** Although `swap()` and `exchange()` may be specialized in the `std` namespace, most recommend to specialize them in the same namespace as their template argument type. The advantage then is that so-called *argument-dependent lookup (ADL)* works. In other words, that '`swap(x,y)`' works without `using` directives or declarations and without specifying the namespace of `swap()`. The ADL rules basically stipulate that a non-member function should be looked up first in the namespace of its arguments. Generic code should then use the following idiom to fall back to `std::swap()` if need be: '`using std::swap; swap(x,y);`'. Simply writing `std::swap(x,y)` would not use user-defined `swap()` functions outside the `std` namespace, while `swap(x,y)` alone would not work *unless* there is such a user-defined function.

■ **Tip** One use case where `std::exchange()` often comes in handy is when implementing move members. For example:

```
class Node {};
class Tree {
   Node* m_root = nullptr;   // Pointer to dynamic memory
 public:
   Tree(Tree&& other) noexcept
      : m_root(std::exchange(other.m_root, nullptr)) {}
   //...
};
```

Pairs and Tuples

Pairs <utility>

The `std::pair<T1,T2>` template `struct` is a copyable, movable, swappable, (lexicographically) comparable `struct` that stores a pair of `T1` and `T2` values in its public `first` and `second` member variables. A default-constructed pair zero-initializes its values, but initial values may be provided as well:

```
std::pair<unsigned int, Person> p(42u, Person("Douglas", "Adams"));
```

The template type arguments can be deduced automatically with C++17 class template argument deduction. The previous definition of `p` can be simplified as follows:

```
std::pair p(42u, Person("Douglas", "Adams"));
```

If your compiler does not yet support class template argument deduction, then you can use a helper function provided by the Standard Library to deduce the types as follows:

```
auto p = std::make_pair(42u, Person("Douglas", "Adams"));
```

Conversely, decomposing a `pair` back into its elements can be done using C++17's structured bindings. For example:

```
auto [number, person] = p;
```

If your compiler does not yet support structured bindings, you can use `std::tie()` instead, as explained in the section on tuples.

■ **Tip** Not all types can be moved efficiently, and would have to be copied when constructing a pair. For bigger objects (e.g., those that contain fixed-size arrays), this could be a performance issue. Other types may even not be copyable at all. For such cases, `std::pair` has a special 'piecewise' constructor to perform in-place construction of its two members. It is called with a special constant, followed by two tuples (see next section) containing the arguments to forward to the constructors of both members.

For instance (`forward_as_tuple()` is used to not copy the strings to a temporary tuple):

```
std::pair<unsigned, Person> p(std::piecewise_construct,
  std::make_tuple(42u), std::forward_as_tuple("Douglas", "Adams"));
```

Piecewise construction can also be used with the `emplace()` functions of the containers in Chapter 3 (these functions are similarly defined to avoid unwanted copying), and in particular with those of `std::map` and `std::unordered_map`. Note that with piecewise construction you have to provide the template type arguments; C++17 class template argument deduction does not work in this case.

Tuples **<tuple>**

`std::tuple` is a generalization of `pair` that allows any number of values to be stored (i.e., zero or more, not just two): `std::tuple<Type...>`. It is mostly analogous to `pair`. The main difference is that the individual values are not stored in public member variables. Instead, you can access them using one of the `get()` template functions:

```
std::tuple t(1, 2, 0.3, std::string("4"));
std::cout << std::get<0>(t) << '\n';        // get using 0-based index
std::get<2>(t) = 3.0;       // no set required: get returns a reference
std::cout << std::get<double>(t) << '\n';    // get using unique type
// std::cout << std::get<int>(t) << '\n';   --> ambiguous: compiler error!
std::string s = std::get<3>(std::move(t));   // move a value out of a tuple
```

C++17 class template argument deduction is again supported as shown in the previous code snippet. Similar as for `std::pair`, there is a helper method `std::make_tuple()` in case you need it. For example:

```
auto t = std::make_tuple(1, 2, 0.3, std::string("4"));
```

Structured bindings from C++17 can be used to obtain the elements stored inside a `tuple`. For example:

```
auto [one, two, three, s] = t;  // = std::move(t) moves string into s
```

Alternatively, you can unpack a `tuple` using the `tie()` function. The special `std::ignore` constant may be used to exclude any value:

```
int one, two; std::string s;
std::tie(one, two, std::ignore,s) = t; //= std::move(t) moves to s
```

■ **Tip** The `std::tie()` function may be used to compactly implement lexicographical comparisons based on multiple values. For instance, the body of `operator<` for our `Person` class in the Introduction could be written as

```
return std::tie(lhs.m_isVIP, lhs.m_lastName, lhs.m_firstName)
     < std::tie(rhs.m_isVIP, rhs.m_lastName, rhs.m_firstName);
```

Two helper structs exist to obtain the size and element types of a given tuple as well, which is mainly useful when writing generic code:

```
std::cout << std::tuple_size<decltype(t)>::value << '\n';      // 4
std::tuple_element<0,decltype(t)>::type one = std::get<0>(t);  // int
```

Note that get(), tuple_size, and tuple_element are also defined for pair and std::array (cf. Chapter 3) in their respective headers, but not tie(). std::tie() does work on a pair though because std::tuple has a converting assignment operator that allows you to assign pairs to tuples (of matching types).

Another helper function for tuples is std::forward_as_tuple(), which creates a tuple of references to its arguments. These are lvalue references generally, but rvalue references are maintained, as with std::forward() explained earlier.

std::apply() invokes a given function or function-like object by passing the individual elements of a given tuple as arguments. A function f(const std::string&, int), for instance, can thus be called as follows C++17:

```
std::tuple tuple("test", 123);
std::apply(f, tuple);
```

Similarly, std::make_from_tuple<T>(t) constructs an object of type T by passing the elements of a given tuple t as arguments to its constructor C++17.

Both apply() and make_from_tuple() do not only work with tuple but also with pair and array. Basically they work with anything that supports std::get<>() and std::tuple_size().

Finally, tuples offer facilities for custom allocators as well, but this is an advanced topic that falls outside the scope of this book.

std::byte C++17

<cstddef>

std::byte is a type to represent a single byte of memory. It can be used to access raw memory. Before C++17, the go-to types for this purpose were char and unsigned char. The difference with byte is that byte is actually a scoped enumeration type:

```
enum class byte : unsigned char {};
```

As such, byte is not implicitly convertible to numerical types such as int and does not support arithmetic operations. Only explicit casts to numerical types are allowed. A function std::to_integer<>() is provided to perform such casts as well.

Only a few operations are allowed on bytes. The <<, >>, <<=, and >>= operators are supported to perform bit shifts, and the |, &, ^, ~, |=, &=, and ^= operators can be used to perform logical operations on bytes.

Here is an example of using `std::byte`:

```
std::byte b1{ 1 }, b2{ 42 };  // More realistic examples would read bytes
std::byte b = b1 | b2;        // from a file (Chapter 5), the network, ...
b <<= 5;
int v1 = static_cast<int>(b);
auto v2 = unsigned int(b);
long v3 = std::to_integer<long>(b);
std::cout << v1 << ' '<< v2 << ' ' << v3 << std::endl; // 96 96 96
```

Relational Operators **<utility>**

A nice set of relational operators is provided in the `std::rel_ops` namespace: `!=`, `<=`, `>`, and `>=`. The first one is implemented in terms of `operator==`, the remaining forward to `operator<`. When you add '`using namespace std::rel_ops`', your class therefore only needs to implement `operator==` and `<`. The other operators are then generated automatically:

```
// Works even though only operator< is defined for our Person class:
using namespace std::rel_ops;
const bool comparison = ( Person("Alexander") > Person("Bob") );
std::cout << comparison;  // 0 (Alexander is not greater)
```

Smart Pointers **<memory>**

A *smart pointer* is an RAII-style object that (typically) decorates and mimics a pointer to heap-allocated memory while guaranteeing this memory is deallocated at all times once appropriate. As a rule, modern C++ programs should never use raw pointers to manage (co-)owned dynamic memory: all dynamic memory should be managed either by a smart pointer or a container such as `vector` (cf. Chapter 3). Consequently, C++ programs should rarely directly call `delete` or `delete[]` anymore. Doing so will go a long way toward preventing memory leaks.

Exclusive Ownership Pointers
std::unique_ptr

A `unique_ptr` has exclusive ownership of a pointer to heap memory and can therefore not be copied, only moved or swapped. Other than that, it mostly behaves like a regular pointer. The following illustrates its basic usage on the stack:

```
{ std::unique_ptr<Person> jeff(new Person("Jeffrey"));
  if (jeff != nullptr) jeff->SetLastName("Griffin");
  if (jeff) DoSomethingWith(*jeff);             // Dereference as Person&
}                    // jeff is deleted, even if DoSomethingWith() throws
```

The -> and * operators ensure a unique_ptr can mostly be used like a raw pointer. Comparison operators ==, !=, <, >, <=, and >= are provided to compare two unique_ptrs, or a unique_ptr with nullptr (in either order), but not for comparing a unique_ptr<T> with a T value. To do the latter, get() must be called to access the raw pointer. A unique_ptr also conveniently casts to a Boolean to check for nullptr.

Construction is facilitated using the helper function make_unique(), for example:

```
{ auto jeff = std::make_unique<Person>("Jeffrey");
  ...
```

Typical uses of unique_ptrs that make them a truly essential utility include the following:

- They can be stored safely inside the containers of Chapter 3.
- When used as member variables of another class, they eliminate the need for explicit deletes in its destructor. Moreover, they prevent the compiler from generating error-prone copy members for objects that are supposed to exclusively own dynamic memory.
- They are the safest and recommended way to transfer exclusive ownership, either by returning a unique_ptr from a function that creates a heap object or by passing one as an argument to a function that accepts further ownership. This has three major advantages:
  a. In both cases, std::move() generally has to be used, making the ownership transfer explicit.
  b. The intended ownership transfer also becomes apparent from the functions' signatures.
  c. It prevents memory leaks.

A unique_ptr can also manage memory allocated with new[]:

```
{ std::unique_ptr<int[]> array(new int[123]); // or make_unique<int[]>(123)
  for (int i = 0; i < 123; ++i) array[i] = i;
  DoSomethingWith(array.get());                    // Pass raw int* pointer
}                  // array is delete[]'d, even if DoSomethingWith() throws
```

For this template specialization, the dereferencing operators * and -> are replaced with an indexed array access operator []. A more powerful and convenient class to manage dynamic arrays, std::vector, is explained later in Chapter 3.

■ **Tip** Normally we recommend you always use `std::make_unique()` for creating a `unique_ptr`. It is often shorter, avoids the use of the `new` keyword (which during code review should trigger a scan for `delete`), and, prior to C++17, used to be safer as well. In rare occasions, though, you could still consider the use of `new []`, as it is one of few ways to allocate a dynamic array of uninitialized values. Both `make_unique<T[]>()` and `vector<T>` zero-initialize fundamental data types or pointers:

```
auto a2 = std::make_unique<int[]>(123);  // 123 times zero
std::unique_ptr<int[]> a1(new int[123]); // 123 uninitialized values
```

Do keep in mind that this really only matters in performance critical code, which is rare. In all other cases, you should stick with `vector` for managing arrays (see Chapter 3).

■ **Note** C++17's class template argument deduction purposely does not work for smart pointers. The reason is that the compiler cannot always properly deduce the required type. Concretely, if you pass a pointer of type `T*` to the constructor, the compiler has no way of knowing whether this points to a single `T` or an array of `T`s.

A `unique_ptr<T>` has two similar members that are often confused: `release()` and `reset(T*=nullptr)`. The former replaces the old stored pointer (if any) with `nullptr`, while the latter replaces it with the given `T*`. The key difference though is that `release()` *does not delete the old pointer*. Instead, `release()` is intended to release ownership of the stored pointer: it simply sets the stored pointer to `nullptr` and returns its old value. This is useful to pass ownership to, for example, a legacy API. `reset()`, on the other hand, is intended to replace the stored pointer with a new value, not necessarily `nullptr`. Before overwriting the old pointer, it is deleted. `reset()` therefore also does not return any value:

```
auto niles = std::make_unique<Person>("Niles", "Crane");
niles.reset(new Person("Niles", "Butler"));      // Niles Crane is deleted
TakeOwnership(niles.release());// TakeOwnership() must delete Niles Butler
```

■ **Tip** Take care for memory leaks when transferring ownership using `release()`. Suppose the preceding example ended with `TakeOwnership(niles.release(), f())`. If the call to `f()` throws after the `unique_ptr` has released ownership, Niles leaks. Therefore, always make sure that expressions containing `release()` subexpressions do not contain any throwing subexpressions as well. In our example, the solution would be to evaluate `f()` on an earlier line, storing its result in a named variable. Transferring using `std::move(niles)`, as recommended earlier, would never leak either by the way. But for legacy APIs this is not always an option.

■ **Caution** A fairly common mistake is to use `release()` where `reset()` was intended, the latter with the default `nullptr` argument, ignoring the value returned by `release()`. The object formerly owned by the `unique_ptr` then leaks, which often goes by unnoticed.

An advanced feature of `unique_ptr`s is that they can use a custom *deleter*. The deleter is the functor that is executed when destroying the owned pointer. This is useful for nondefault memory allocation, to do additional cleanup or, for example, to manage a file pointer as returned by the C function `fopen()` (defined in `<cstdio>`):

```
{ std::unique_ptr<FILE, std::function<void(FILE*)>>
      smartFilePtr(fopen("test.txt", "r"), fclose);
  DoSomethingWith(smartFilePtr.get());
}              // The FILE* is closed, even if DoSomethingWith() throws
```

This example uses a deleter of type `std::function` (defined in the `<functional>` header, discussed later in this chapter) initialized with a function pointer, but any function pointer or functor type may be used instead.

■ **Tip** You can also use `decltype()` to specify the type of the custom deletion function, as in, for instance, `std::unique_ptr<FILE, decltype(&fclose)>`.

std::auto_ptr

In pre-C++17, the `<memory>` header defined a second smart pointer type for exclusive ownership, namely, `std::auto_ptr`. In C++17, this has been removed in favor of `unique_ptr`. Essentially, an `auto_ptr` was a flawed `unique_ptr` that is implicitly moved when copied: this not only makes them error-prone but also dangerous (and in fact illegal) to use with the standard containers and algorithms of Chapters 3 and 4.

Shared Ownership Pointers

std::shared_ptr

When multiple entities share the same heap-allocated object, it is not always obvious or possible to assign a single owner to it. For such cases, `shared_ptr`s exist, defined in `<memory>`. These smart pointers maintain a thread-safe reference count for a shared memory resource, which is deleted once its reference count reaches zero, that is, once the last `shared_ptr` that co-owned it is destructed. The `use_count()` member returns the reference count, and `unique()`[1] checks whether the count equals one.

[1] `unique()` is deprecated in C++17 and removed in C++20.

Like a unique_ptr, it has ->, *, cast-to-Boolean, and comparison operators to mimic a raw pointer. Equivalent get() and reset() members are provided as well, but no release(). Also similar to a unique_ptr, a shared_ptr can manage dynamic arrays, but only since C++17. What really sets shared_ptr apart from unique_ptrs though is that they can and are intended to be copied:

```
{ auto bond = std::make_shared<int>(007);       // bond.use_count() == 1
  auto james = bond;  // james.use_count() == 2 && bond.use_count() == 2
  bond.reset();       // james.use_count() == 1 && bond.use_count() == 0
}                     // 007 is deleted
```

A shared_ptr can be constructed by moving a unique_ptr into it, but not the other way around:

```
auto muhammed = std::make_unique<Person>("Muhammed");
std::shared_ptr<Person> mountain = std::move(muhammed);
//muhammed = std::move(mountain); // mountain cannot be moved to Muhammed
```

To construct a new shared_ptr, it is again recommended to use make_shared(): for the same reason as with make_unique() (shorter code), but in this case also because it is more efficient.

Custom deleters are again supported. Unlike with unique_ptr though, the deleter's type is not a type argument of the shared_ptr template. The declaration analogous to the one in our earlier example thus becomes

```
std::shared_ptr<FILE> smartFilePtr(fopen("test.txt", "r"), fclose);
```

To obtain a shared_ptr to a related type, use std::static_pointer_cast(), dynamic_pointer_cast(), const_pointer_cast(), or reinterpret_pointer_cast() (reinterpret_pointer_cast() only since C++17). If the result is non-null, the reference count shall safely be incremented with one. An example will clarify:

```
class A { public: virtual ~A() {}; };
class B : public A {};
class C {};

std::shared_ptr<A> a = std::make_shared<B>();    // a points to a new B()
std::shared_ptr<B> b = std::dynamic_pointer_cast<B>(a);
std::shared_ptr<C> c = std::dynamic_pointer_cast<C>(a);
// (c == nullptr) && (a.use_count() == b.use_count() == 2)
```

A lesser-known feature of shared_ptrs is called *aliasing* and is used for sharing parts of an already shared object. It is best introduced with an example:

```
struct A { int* member; /* ... */ };
auto a = std::make_shared<A>();
auto m = std::shared_ptr<int>(a, a->member);  // aliasing constructor
// a.use_count() == m.use_count() == 2
```

A shared_ptr has both an *owned* pointer and a *stored* pointer. The former determines the reference counting, the latter is returned by get(), *, and ->. Generally both are the same, but not if constructed with the aliasing constructor. Almost all operations use the stored pointer, including the comparison operators <, >=, and so on. To compare based on the owned rather than the stored pointer, use the owner_before() member or std::owner_less<T=void> functor class (functors are explained shortly). This is useful, for example, when storing shared_ptrs in a std::set (see Chapter 3).

std::weak_ptr

There are times, particularly when building caches of shared objects, that you want to keep a reference to a shared object should you need it, but you do not want your reference to necessarily prevent the deletion of the object. This concept is commonly called a *weak reference* and is offered by <memory> in the form of a std::weak_ptr.

A nonempty weak_ptr is constructed with a shared_ptr or results from assigning a shared_ptr to it afterward. These pointers can again be freely copied, moved, or swapped. While a weak_ptr does not co-own the resource, it can access its use_count(). To check whether the shared resource still exists, expired() can be used as well (it is equivalent to use_count()==0). The weak_ptr, however, does not have direct access to the shared raw pointer, as nothing would then prevent the last co-owner from concurrently deleting it. To access the resource, a weak_ptr first has to be promoted to a co-owning shared_ptr using its lock() member:

```
auto s = std::make_shared<std::string>("SharedString");
auto w = std::weak_ptr<std::string>(s);//w.use_count() == s.use_count() == 1
{ std::shared_ptr<std::string> s2 = w.lock();
  DoSomethingWith(*s2);               // w.use_count() == s.use_count() == 2
}                                     // w.use_count() == s.use_count() == 1
s.reset();                            // w.expired() == true
```

std::enable_shared_from_this

The Standard Library provides std::enable_shared_from_this<T>, a mixin class template that allows objects to safely return a shared_ptr<T> to themselves. When a class inherits from enable_shared_from_this<T>, it gets the following extra methods:

- **shared_from_this()** returns a shared_ptr<T> to this.
- **weak_from_this()** returns a weak_ptr<T> to this.

There is one important caveat though when using this template.

■ **Caution** `shared_from_this()` can only be used when a pointer to `this` has already been stored in a `shared_ptr` before; otherwise, `shared_from_this()` throws a `std::bad_weak_ptr` exception. `weak_from_this()` simply returns an empty `weak_ptr` if no pointer to the object has been placed inside a `shared_ptr` yet.

In the following example, the pointers `p1`, `p2`, and `p3` point to the same instance of `MyObject`:

```
class MyObject : public std::enable_shared_from_this<MyObject> {
 public:
   std::shared_ptr<MyObject> pointer() { return shared_from_this(); }
};

int main() {
   auto p1 = std::make_shared<MyObject>();
   auto p2 = p1->pointer();
   auto p3 = p1->shared_from_this();

   MyObject unshared;
   auto p4 = unshared.shared_from_this (); // Throws std::bad_weak_ptr
}
```

Function Objects <functional>

A *function object* or *functor* is an object with an `operator()(T1,...,Tn)` (`n` may be zero), allowing it to be invoked just like a function or operator. Here is an example:

```
template <typename T> struct my_plus
   { T operator() (const T& x, const T& y) const {return x+y;} };
my_plus<int> functor;
std::cout << functor(11,22) << std::endl;                    // 33
```

Functors can not only be passed to many standard algorithms (Chapter 4) and concurrency constructs (Chapter 7) but are also very useful for creating your own generic algorithms or, for example, for storing or providing callback functions.

This section outlines the functors defined in `<functional>`, as well as its facilities for creating and working with functors.[2]

---

[2] The old `ptr_fun()`, `mem_fun()`, `mem_fun_ref()`, `bind1st()`, `bind2nd()`, their return types, as well as the base classes `unary_function` and `binary_function` that were defined in `<functional>` have been removed from the C++17 version of the standard.

Before we delve into functors though, first a short word on the reference wrapper utilities that are defined in the <functional> header.

■ **Note** In the text that follows, the term *callable* is used. A callable is something that can be invoked—such as a function, a function pointer, a function object, a closure resulting from a lambda expression, and so on.

Reference Wrappers

The functions `std::ref()` and `cref()` return `std::reference_wrapper<T>` instances that simply wrap a (`const`) `T&` reference to their input argument. This reference can then be extracted explicitly with `get()` or implicitly by casting to `T&`.

As these wrappers can safely be copied, they can be used, for example, to pass references to template functions that take their arguments by value, badly forward their arguments (forwarding is discussed earlier in this chapter), or simply copy their arguments for other reasons. Standard template functions that do not accept references as arguments, but do work with `ref()`/`cref()`, include `std::thread()` and `async()` (see Chapter 7) and the `std::bind()` function discussed shortly.

These wrappers can be assigned to as well, thus enabling storing references into the containers of Chapter 3. In the following example, for instance, one could not have declared a `vector<int&>`, because `int&` values cannot be assigned to:

```
int i = 234;
std::vector v{ std::ref(i) }; // cf. Chapter 3
// Or: std::vector<std::reference_wrapper<int>> v{ std::ref(i) };
v[0].get() = 432;  // Occasionally, like here, an explicit get() is needed
                   // (v[0] returns reference_wrapper<int>&, not int&).
std::cout << v[0] << "==" << i << std::endl;              // 432==432
```

Predefined Functors

The <functional> header provides an entire series of functor `structs` similar to the `my_plus` example used earlier in the introduction of this section:

- `plus`, `minus`, `multiplies`, `divides`, `modulus`, and `negate`
- `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal`, and `less_equal`
- `logical_and`, `logical_or`, and `logical_not`
- `bit_and`, `bit_or`, `bit_xor`, and `bit_not`

These functors often result in short, readable code, even more so than with lambda expressions. The following example sorts an array in descending order (with the default being ascending) using the `sort()` algorithm explained in Chapter 4:

```
int array[] = { 7, 9, 7, 2, 0, 4 };
std::sort(std::begin(array), std::end(array), std::greater<int>());
```

As of C++14, all preceding functor classes have a special specialization for `T` equal to `void`, and `void` has also become the default template type argument. These are called *transparent operator functors*, as their function call operator conveniently deduces the parameter type. In the preceding `sort()` example, one could simply use `std::greater<>`. The same functor can even be used for different types:

```
std::plus<> fun;                        // defaults to std::plus<void>
std::cout << fun(234,432) << ' ' << fun(1.101,2.0405) << std::endl;
```

As Chapter 3 explains, the transparent `std::less<>` and `greater<>` functors are also the preferred comparison functors for ordered associative containers.

Binding Function Arguments

The `std::bind()` function may be used to wrap a copy of any callable while changing its signature: parameters may be reordered, assigned fixed values, and so on. To specify which arguments to forward to the wrapped callable, a sequence of either values or so-called placeholders (`_1`, `_2`, etc.) is passed to `bind()`. The first argument passed to the bound functor is forwarded to all occurrences of placeholder `_1`, the second to those of `_2`, and so on. The maximum number of available placeholders is implementation specific. Some examples to clarify:

```
bool my_less(int x, int y) { return x < y; }
void f(std::string& x, char y) { x += y; }
int main() {
   using namespace std::placeholders;         // contains _1, _2, _3, ...
   auto my_greater = std::bind(my_less, _2, _1); // function + swap _1 _2
   auto twice = std::bind(std::plus<int>{}, _1, _1);// functor + twice _1
   auto plus5 = std::bind(std::plus<int>{}, _1, 5); // functor + fixed 5
   std::cout << my_greater(twice(13), plus5(20)) << '\n';    // 1 (true)

   // bind() expressions may be nested whilst sharing placeholders:
   auto g = std::bind(my_greater,std::bind(twice,_1),std::bind(plus5,_1));
   std::cout << g(10) << ' ' << g(4) <<'\n';         // 1 0 (true false)
```

```
  // Use std::ref()/cref() to pass references
  // (For containers, algorithms, and strings seechapters 3, 4, and 6)
  std::vector v{ 'c', 'o', 'n', 'c', 'a', 't' };
  std::string concat;
  std::for_each(begin(v), end(v), std::bind(f, std::ref(concat), _1));
  std::cout << concat << std::endl;
}
```

The type of the functor that is returned by `std::bind()` is unspecified, much like the type of a lambda expression (also a functor).

■ **Tip** As of C++14, there is nothing `std::bind()` can do that lambda expressions cannot do as well. As lambda expressions are mostly easier to write and read than `bind()` expressions, we therefore do not recommend using `std::bind()` too often.

Negating a Callable C++17

Passing a callable, `predicate`, to `std::not_fn()` creates and returns a new functor (of an unspecified type) that negates `predicate`'s result (i.e., evaluates to `!predicate()`). This new functor has the same parameter types as the original callable.

Prior to C++17, you had to resort to `std::not1()` / `not2()`, which only worked for unary/binary functors that satisfied certain requirements. Both `not1()` and `not2()` are deprecated since C++17 and are not further discussed.

Generic Function Wrappers

The `std::function` class template is designed for wrapping a copy of any kind of *callable*. This includes functors of undefined types, such as the results of `std::bind()`, `not_fn()`, and lambda expressions:

```
bool my_less(int x, int y) { return x < y; }
int main() {
   std::function<bool(int,int)> test = my_less;       // function
   test = &my_less;                                   // function pointer
   test = std::less<>{};                              // function object
   test = [](int x, int y) {return x < y; };          // lambda closure
   if (test) std::cout << test(234,432) << std::endl; // 1 (true)
}
```

If a default-constructed `function` object is called, a `std::bad_function_call` exception is thrown. To verify whether a `function` may be called, it conveniently casts to a

Boolean. Alternatively, you may compare a `function` to a `nullptr` using == or !=, just like you would with a function pointer.

Other members include `target<Type>()` to obtain a pointer to the wrapped entity (the correct Type must be specified, otherwise the member returns `nullptr`) and `target_type()` which returns the `type_info` for this wrapped entity (`type_info` is explained under "Type Utilities" later in this chapter).

■ **Tip** A lesser-known feature of `std::ref()`, `cref()`, and their return type `reference_wrapper` seen earlier is that they can also be used to wrap callables. Unlike a `std::function` though, which stores a *copy* of the callable, a `reference_wrapper` stores a *reference* to it. This is useful when passing a functor you do not want to be copied—for example, because it is too large (performance), stateful, or simply uncopyable—to an algorithm that accepts it or may pass it around by value. For example:

```
function_that_copies_its_callable_argument(std::ref(my_functor));
```

Note that for the standard algorithms of Chapter 4, it is generally unspecified how often they copy their arguments. So to guarantee no copies are made, one must use `(c)ref()`.

Functors for Class Members

Both `std::function` and `bind()` may be used to create functors that evaluate to a given object's member variable or that call a member function on a given object. A third option is to use `std::mem_fn()`, which is intended specifically for this purpose:

```
struct my_struct { int val; bool fun(int i) { return val == i; } };
int main() {
   my_struct s{234};

   std::function<int(my_struct&)> f_get_val = &my_struct::val;
   std::function<bool(my_struct&,int)> f_call_fun = &my_struct::fun;
   std::cout << f_get_val(s) << ' ' << f_call_fun(s, 123) << std::endl;

   using std::placeholders::_1;
   auto b_get_val = std::bind(&my_struct::val, _1);
   auto b_call_fun_on_s = std::bind(&my_struct::fun, std::ref(s), _1);
   std::cout << b_get_val(s) << ' ' << b_call_fun_on_s(234) << std::endl;

   auto m_get_val = std::mem_fn(&my_struct::val);
   auto m_call_fun = std::mem_fn(&my_struct::fun);
   std::cout << m_get_val(s) << ' ' << m_call_fun(s, 456) << std::endl;
}
```

The member functors created by either `bind()` or `mem_fn()`, but not `std::functions`, may also be called with a pointer or one of the standard smart pointers (cf. previous section) as the first argument (i.e., without dereferencing). Interesting also about the `bind()` option is that it can bind the target object itself (cf. `b_call_fun_on_s`). If that is not required, `std::mem_fn()` generally results in the shortest code as it deduces the entire type. A more realistic example is this (`vector`, `count_if()`, and `string` are explained in Chapters 3, 4, and 6, respectively):

```
std::vector<std::string> v{ "Test", "", "123", "", "" };
std::cout <<
   std::count_if(begin(v), end(v), std::mem_fn(&std::string::empty)); // 3
```

Initializer Lists **<initializer_list>**

The `initializer_list<T>` type is used by the C++ compiler to represent the result of initializer-list declarations:

```
auto list = { 1, 2, 3 };    // list has type std::initializer_list<int>
```

This curly braces syntax is the only way to create nonempty initializer lists. Once created, `initializer_lists` are immutable. Their few operations, `size()`, `begin()`, and `end()`, are analogous to those of containers (Chapter 3). When constructing an `initializer_list` from a list of initialization values, the list stores a copy of those values. However, copying an `initializer_list` does not copy the elements: the new copy simply refers to the same array of values.

The single most common use case for `initializer_lists` is probably *initializer-list constructors*, which are special in the sense that they take precedence over any other constructors when curly braces are used:

```
class ExampleClass {
 public:
   ExampleClass(int, int) { /* ... */ };
   ExampleClass(std::initializer_list<int>) { /* ... */};
};
ExampleClass a(1, 2);   // (int, int) constructor is used
ExampleClass b{1, 2};   // initializer_list<int> constructor is used
```

All container classes of Chapter 3, for instance, have initializer-list constructors to initialize them with a list of values.

Vocabulary Types C++17

std::optional <optional>

A `std::optional<T>` can either hold a value of type `T` or not. Use cases include optional return values from functions, optional function parameters, and optional class data members. Using `std::optional` removes the need to resort to sentinel values (e.g., `-1`, `EOF`, `nullptr`, `NaN`, etc.) to represent, for instance, error conditions or values that are not (yet) set. It also clearly expresses what to expect: a value that may not always be present. The following code snippet shows how an `optional` can be used as a return type of a function:

```
std::optional<double> divide(double n, double d) {
   if (d != 0.0) return n / d;
   else return {};
}

int main() {
   auto result = divide(12, 3);
   if (result.has_value())                          // or: if (result)
      std::cout << "result is: " << result.value();  // or: ... << *result
}
```

In the preceding example, the difference between `result.value()` and `*result` is that for an empty `optional` the former throws a `bad_optional_access` exception, whereas the latter results in undefined behavior.

The standard requires an `optional` to hold its value in place, so when creating an instance of `optional<T>`, it is made big enough to hold such a `T`.

std::nullopt_t

The Standard Library provides the type `std::nullopt_t` and a global instance of that type called `std::nullopt`. A `nullopt_t` can be passed to the constructor of `optional` to create an `optional` without a value. It can also be used in assignment statements to remove any value currently in an `optional`, and in comparisons to verify whether an `optional` has a value or not.

The `divide()` example from earlier can be written as follows:

```
std::optional<double> divide(double n, double d) {
   if (d != 0.0) return n / d;
   else return std::nullopt;
}
```

```
int main() {
   auto result = divide(12, 3);
   if (result != std::nullopt)
      std::cout << "result is: " << result.value();
}
```

Constructors

An optional can be constructed either from a given T value or from U or optional<U> if U is convertible to T. Default-constructed optionals and optionals constructed from a nullopt_t have no value. A std::optional<T> is copyable and/or movable if T is copyable and/or movable.

Finally, optional also supports the following in-place constructor:

```
optional(in_place_t, [initializer_list,] Args&&... args)
```

This creates an optional holding a T value constructed in place by forwarding the optional initializer list and arguments to the appropriate constructor of type T. The standard provides std::in_place which is of type std::in_place_t, and which you can use to pass to this constructor. This in-place construction works in a similar fashion as the in-place construction of pairs. Here is a quick example:

```
auto opt = std::optional<std::string>(std::in_place, "Hello");
```

A helper function to create optionals is provided as well, std::make_optional(). It comes in the following flavors:

- make_optional(T&& value): Creates an optional from the given value
- make_optional([initializer_list,] Args&&... args): Creates an optional with a value constructed in place by calling a constructor of T and passing it the optional initializer list and forward<Args>(args)...

■ **Note** References cannot be stored as is in an optional. You can however create an optional<reference_wrapper<T>> or optional<reference_wrapper<const T>>. Reference wrappers are discussed earlier in this chapter in the section on function objects.

Methods and Non-member Functions

The following methods are provided:

Method	Description
`operator bool()` `bool has_value()`	Returns `true` if the `optional` contains a value, `false` otherwise.
`T* operator->()` `T& operator*()`	Returns a pointer or a reference to the wrapped value. The behavior is undefined if the `optional` has no value.
`T& value()`	Returns a *reference* to the wrapped value. Throws `bad_optional_access` if the optional has no value.
`T value_or(U&& default_value)`	Returns a *copy* of the wrapped value if any; otherwise returns the given default value.
`operator=()`	Assigns a value or `nullopt_t` to the `optional`.
`T& emplace(Args&&... args)` `T& emplace(initializer_list, Args&&... args)`	Constructs a value for the `optional` in place by forwarding the given initializer list (if any) and arguments to a constructor of `T`.
`reset()`	Empties the `optional` by destroying the wrapped value. No new value can be passed to `reset()` (this is unlike the `reset()` of smart pointers earlier).
`swap(optional&)`	Swaps the contents with another `optional`.

Three sets of comparison operators (==, !=, <, >, <=, and >=) are provided as non-member functions: comparing two `optional` instances, comparing an `optional` with `std::nullopt`, and comparing an `optional` with a value. A non-member `std::swap(optional&)` is also available.

std::variant <variant>

An instance of `std::variant` holds a value of one of a predefined set of types, called *alternatives*. You can think of it as a type-safe union. For example, `variant<int, long, string>` is a `variant` that holds an instance of one of the three specified alternatives at any given time: either an `int`, a `long`, or a `string`.

Similar to an `optional`, a `variant` is required to store its value in place, so it is made big enough to hold an instance of the biggest specified alternative type. Also similar to `optional`, a `variant` cannot store references as is. If you need to store references, use reference wrappers (discussed in the section on function objects). Unlike `optional`, however, a `variant` normally always holds a value.

Construction

Default Construction

A variant is never empty, unless an exception is thrown while constructing the value instance to store in the variant. The default constructor of a variant does not create an 'empty' variant, but one that holds a default-constructed value of the first supported alternative type. The compiler will raise an error if you try to default-construct a variant for which the first alternative is not default-constructible. For example, the following code does not compile:

```
class NoDefaultCtor {
 public:
   NoDefaultCtor() = delete;
   NoDefaultCtor(int) {}
};

std::variant<NoDefaultCtor> v;
```

To make it possible to use variant with types that are not default-constructible as the first alternative, the <variant> header provides std::monostate, a trivial default-constructible type. You use it as follows:

```
std::variant<std::monostate, NoDefaultCtor> v;
```

Constructors

The following constructors are available:

- variant(): Constructs a variant containing a value-initialized instance of the variant's first alternative.
- Copy and move constructors.
- variant(T&& t): Constructs a variant containing a value of one of the variant's alternative types which can be constructed by forwarding the given value t to a constructor of that type.
- variant(in_place_type_t<T>, [initializer_list,] Args&&... args): Constructs a variant containing a value of type T constructed by calling a constructor of T and passing it the optional initializer list and forward<Args>(args)...
- variant(in_place_index_t<I>, [initializer_list,] Args&&... args): Constructs a variant containing a value of the type of the I^{th} alternative, constructed by calling a constructor of that type and passing it the optional initializer list and forward<Args>(args)...

std::in_place_type_t<T> and in_place_index_t<I> are tags to be able to differentiate overloaded constructors. The standard also provides std::in_place_type<T> and in_place_index<I> which are predefined instances of the corresponding

_t types, and which you can pass as arguments to the constructors. Here is an example of using one of the in-place constructors:

```
std::variant<int, std::string> v(std::in_place_type<std::string>, "Hello");
```

Methods and Non-member Functions

A std::variant supports the following methods:

Method	Description
valueless_by_exception()	Returns true if the variant is empty. This state can only be reached when an exception is thrown while constructing the value instance to store in the variant.
index()	Returns the index of the type (alternative) of the value currently stored in the variant, or std::variant_npos for a valueless variant.
operator=()	Assigns a value to the variant.
emplace<T>() emplace<I>()	Constructs a new value in place. It has a similar set of overloads as the constructors, i.e., it can construct a value based on a given type T or on the index I of the alternative type, and it either accepts a set of arguments or an initializer list and a set of arguments. The return value is a reference to the created value.
swap(variant&)	Swaps the contents with another variant.

Additionally, the Standard Library provides a set of comparison operators (==, !=, <, >, <=, and >=) as non-member functions and includes the following non-member helper functions to work with variants:

Non-member Function	Description
holds_alternative<T> (const variant&)	Returns true if the given variant holds a value of the specified type T.
get<T>(variant&) get<I>(variant&) get<T>(const variant&) get<I>(const variant&) get<T>(variant&&) get<I>(variant&&)	Retrieves the value currently held by a variant, either based on the type or the index of the desired alternative. Returns a reference to the requested value and throws std::bad_variant_access if the current variant does not hold an alternative of the requested type or index. The behavior is unspecified if either the requested type is not unique among the alternatives or the given index is not a valid index for the variant.

(*continued*)

Non-member Function	Description

`get_if<T>(variant*)`
`get_if<I>(variant*)`
`get_if<T>(const variant*)`
`get_if<I>(const variant*)`
	Nonthrowing versions of the `get<>()` functions that accept and return pointers instead of references. Instead of throwing, these return `nullptr` on errors.

`swap(variant&)`	Swaps the contents with another `variant`.

Example Usage

Here is a quick example of how to use the `variant` type:

```
std::variant<int, std::string> v1; // Value-constructed variant contains
                                   // a value-initialized value of the
                                   // first alternative type (int).
std::cout << std::holds_alternative<int>(v1) << std::endl;  // 1 (true)
std::cout << v1.index() << std::endl;                       // 0

v1 = "In manufacturing, we try to stamp out variance. "
     "With people, variance is everything.";

std::cout << std::get<1>(v1) << std::endl;                  // In manu...
std::cout << *std::get_if<std::string>(&v1) << std::endl;   // In manu...
```

Visitation

A final helper function provided is `std::visit()`:

```
template<class Visitor, class... Variants>
constexpr auto visit(Visitor&& visitor, Variants&&... v);
```

This function accepts a visitor and one or more `variants` to be visited. The return type is the return type of the visitor that was invoked. If `visit()` is called with one `variant`, then the given visitor will be invoked with a single argument: the value of the given `variant`. A `std::bad_variant_access` exception is thrown if the given `variant` is empty.

Here is an example of using `visit()` to visit a single `variant`:

```
std::variant<std::string, int> v1 = { "Hello world!" }, v2 = 42;

// A simple visitor.
auto myVisitor = [](auto&& value) { std::cout << value << std::endl; };
std::visit(myVisitor, v1);  // Hello world!
std::visit(myVisitor, v2);  // 42
```

```
// A type-matching visitor to handle specific types.
struct IsNullVisitor {
  bool operator()(const std::string& s) const { return s.empty(); }
  bool operator()(int i) const { return i == 0; }
};

std::cout << std::visit(IsNullVisitor{}, v1) << std::endl; // 0
std::cout << std::visit(IsNullVisitor{}, v2) << std::endl; // 0
```

`visit()` can also visit multiple `variants` at once. The result of calling `visit()` with multiple `variants` is a single invocation of the given visitor passing as many arguments as there are `variant` arguments. For example, if `visit()` is called with two `variants`, then the given visitor will be invoked with two arguments: the value of the first given `variant`, followed by the value of the second given `variant`. A `std::bad_variant_access` exception is thrown if any of the `variants` is empty. A type-matching visitor should have a handler for all combinations of the alternative types in the `variants`. Here is an example of a type-matching visitor visiting two `variants`:

```
class MyVisitor {
 public:
   auto operator()(int, int) const { return "Two ints.";}
   auto operator()(const std::string&, const std::string&) const {
      return "Two strings."; }
   auto operator()(const std::string&, int) const {
      return "A string and an int."; }
   auto operator()(int, const std::string&) const {
      return "An int and a string."; }
};

// A string and an int.
std::cout << std::visit(MyVisitor{}, v1, v2) << std::endl;
```

Helper Classes

The standard provides a `std::variant_size<T>` class template to query, at compile time, the number of alternatives of a given `variant<Ts...>` type—that is, to query, at compile time, the length of its parameter pack `Ts`. This number can be retrieved either with `std::variant_size<T>::value` or with the more convenient variable template `std::variant_size_v<T>` (which is defined exactly as `std::variant_size<T>::value`).

Finally, with `std::variant_alternative<I, T>` you can query the type of the I^{th} alternative type of a given `variant` type. This type can be retrieved with either `std::variant_alternative<I, T>::type` or the more convenient template alias `std::variant_alternative_t<I, T>`.

std::any <any>

An instance of `std::any` can hold a value of any type, or no value at all. As it is capable of holding any kind of type, there is no way of knowing how big an `any` instance must be, so values are usually not stored in place (except when small object optimization kicks in). `any` can be thought of as a type-safe `void*`.

Constructors

The following constructors are available:

- `any()`: Constructs an instance without a value.
- Copy and move constructors.
- `any(T&& value)`: Constructs an `any` containing a value of type `T`, constructed by forwarding the given `value` to the constructor of `T`.
- `any(in_place_type_t<T>, [initializer_list,] Args&&...)`: Constructs an `any` instance containing a value of type `T` constructed in place by forwarding the optional initializer list and arguments to a constructor of `T`. The standard provides `std::in_place_type<T>` which is of type `in_place_type_t<T>`, and which you can use to pass to this constructor.

Finally, the `std::make_any<T>([initializer_list,] Args&&...)` helper functions are provided to create `any` instances, which behave the same as the in-place constructors.

Methods and Non-member Functions

The following methods are provided:

Method	Description
`bool has_value()`	Returns `true` if the `any` contains a value, `false` otherwise.
`type_info& type()`	Returns a `type_info`[3] for the type of the value contained in the `any` instance. If the `any` is empty, returns a `type_info` for `void`.
`operator=()`	Assigns a value to the `any`.
`V& emplace(Args&&... args)` `V& emplace(initializer_list, Args&&... args)`	Constructs a value for the `any` in place by forwarding the given initializer list (if any) and arguments to a constructor of `V`.

(*continued*)

[3] `type_info` is a structure containing information about a given type. It is explained under "Type Utilities" later in this chapter. For now it suffices to know that you can call `name()` on a `type_info` instance to get the full name of a given type. It requires the `<typeinfo>` header.

Method	Description
`reset()`	Empties the `any` by destroying the contained value. No new value can be passed as an argument (this is unlike the `reset()` method of smart pointers).
`swap(any&)`	Swaps the contents with another `any`.

To get type-safe access to the contained value of an `any` instance, use `std::any_cast<T>()`. This function comes in the following five overloads. The last column describes what happens if the value's type of the `any` instance does not match the given type `T`.

Overload	Return Type	On Error
`any_cast<T>(const any&)`	`T`	Throws `bad_any_cast`
`any_cast<T>(any&)`	`T`	Throws `bad_any_cast`
`any_cast<T>(any&&)`	`T`	Throws `bad_any_cast`
`any_cast<T>(const any*)`	`const T*`	Returns `nullptr`
`any_cast<T>(any*)`	`T*`	Returns `nullptr`

Example Usage

Here is a brief example on how to work with `any` instances[4]:

```
std::any a;
std::cout << a.has_value() << std::endl;                // 0
a = 42;
std::cout << a.type().name() << std::endl;              // int
a = "Any sufficiently advanced technology is indistinguishable from magic";
std::cout << a.type().name() << std::endl;              // const char*
std::cout << std::any_cast<const char*>(a) << std::endl; // Any suffi...
a.reset();
std::cout << a.type().name() << std::endl;              // void
```

Date and Time Utilities

<chrono>

The `<chrono>` library introduces utilities mainly for tracking time and durations at varying degrees of precision, determined by the type of *clock* used. To work with dates, one has to use the C-style date and time types and functions defined in `<ctime>`. The `system_clock` from `<chrono>` allows for interoperability with `<ctime>`.

[4] As we explain later, the actual output of `std::type_info::name()` depends on your compiler. With Microsoft Visual Studio 2017, for instance, the output for the type `const char*` is `"char const * __ptr64"`; with GCC 9 it is `"PKc"` (a mangled type name, short for `"Pointer to Konst char"`).

Durations

A `std::chrono::duration<Rep, Period=std::ratio<1>>` expresses a time span as a *tick count*, represented as a `Rep` value which is obtainable through `count()` (`Rep` is or emulates an arithmetic type). The time between two consecutive ticks, or *period*, is statically determined by `Period`, a `std::ratio` type denoting a number (or fraction) of seconds (`std::ratio` is explained in Chapter 1). The default `Period` is 1 second:

```
using namespace std::chrono;
using hours_t = duration<int, std::ratio<3600>>;
using milli_t = duration<int64_t, std::milli>;   // milli==ratio<1,1000>
const hours_t one_hour(1);
const milli_t ms(one_hour);
std::cout << "1h = " << ms.count()<< "ms";       // 1h = 3600000ms
```

The `duration` constructor can convert between `durations` of a different `Period` and/or count `Representation`, as long as no truncation is required. The `duration_cast()` function can be used for truncating conversions as well:

```
// const hours_t back_to_hours(ms);  <-- error (int64_t would be truncated)
const auto back_to_hours = duration_cast<hours_t>(ms);
```

For convenience, several type aliases analogous to those in the previous example are predefined in the `std::chrono` namespace: `hours`, `minutes`, `seconds`, `milliseconds`, `microseconds`, and `nanoseconds`. Each uses an unspecified signed integral `Rep` type, at least big enough to represent a duration of about 1000 years (`Rep` has at least 23, 29, 35, 45, 55, and 64 bits, respectively).

For further convenience, the namespace `std::literals::chrono_literals` contains literal operators to easily create instances of such `duration` types: `h`, `min`, `s`, `ms`, `us`, and `ns`, respectively. They are also made available with a `'using namespace std::chrono'` declaration. When applied on a floating-point literal, the result shall have an unspecified floating-point type as `Rep`:

```
const auto secs = duration_cast<seconds>(0.5h);
std::cout << "0.5h = " << secs.count() << "s";    // 0.5h = 1800s
```

All arithmetic and comparison operators one would intuitively expect for working with `durations` are supported: `+`, `-`, `*`, `/`, `%`, `+=`, `-=`, `*=`, `/=`, `%=`, `++`, `--`, `==`, `!=`, `<`, `>`, `<=`, and `>=`. The following expression, for example, evaluates to a `duration` with `count()` == 22:

```
duration_cast<minutes>((12min + .5h) / 2 + (100ns >= 1ms? -3h : ++59s))
```

C++17 provides `std::chrono::floor()`, `ceil()`, `round()`, and `abs()` that work on `durations`. These functions, except `abs()`, require you to specify a template type argument representing the desired `duration` of the result. Here is an example on how to use `floor()` and `abs()` C++17:

```
auto s1 = -4.2s;                                          // -4.2s
auto s2 = std::chrono::floor<std::chrono::seconds>(s1); // -5s
auto s3 = std::chrono::abs(s1);                           // 4.2s
```

Time Points

A `std::chrono::time_point<Clock, Duration=Clock::duration>` represents a point in time, expressed as a `Duration` since a `Clock`'s *epoch*. This `Duration` may be obtained from its `time_since_epoch()` member. The epoch is defined as the instant in time chosen as the origin for a particular clock, the reference point from which time is measured. The available standard `Clocks` are introduced in the next section.

A `time_point` is generally originally obtained from a member of its `Clock`'s class. It may be constructed from a given `Duration` as well though. If default-constructed, it represents the `Clock`'s epoch. Several arithmetic (`+`, `-`, `+=`, `-=`) and comparison (`==`, `!=`, `<`, `>`, `<=`, `>=`) are again available. Subtracting two `time_points` results in a `Duration`, and `Durations` may be added to and subtracted from a `time_point`. Adding `time_points` together though is not allowed, nor is subtracting one from a `Duration`:

```
using namespace std::chrono;
time_point<system_clock, hours> one_hour(1h);     // 1h since epoch
time_point<system_clock, minutes> sixty_minutes = one_hour;
std::cout << (one_hour - sixty_minutes).count() << std::endl;  // 0
```

Conversion between `time_points` with different `Duration` types works analogously to the conversion of `durations`: implicit conversions are allowed, as long as no truncation is required; otherwise, `time_point_cast()` can be used:

```
auto one_hour = time_point_cast<hours>(sixty_minutes);
```

C++17 provides `std::chrono::floor()`, `ceil()`, and `round()` that work on `timepoints`. These functions require you to specify a template type argument representing the desired `duration` of the result. Here is a quick example C++17:

```
time_point<system_clock, duration<double>> tp1(2.2s);
auto tp2 = std::chrono::floor<seconds>(tp1);      // 2s since epoch
```

Clocks

The `std::chrono` namespace offers three clock types: `steady_clock`, `system_clock`, and `high_resolution_clock`. All clocks define the following static members:

- `now()`: A function returning the current point in time.
- `rep`, `period`, `duration`, `time_point`: Implementation-specific types. `time_point` is the type returned by `now()`: an instantiation of `std::chrono::time_point` with `Duration` type argument equal to `duration`, which in turn equals `std::chrono::duration<rep, period>`.
- `is_steady`: A Boolean constant which is `true` if the time between clock ticks is constant and two consecutive calls to `now()` always return `time_points` `t1` and `t2` for which `t1 <= t2`.

The only clock that is guaranteed to be steady is `steady_clock`. That is, this clock cannot be adjusted. The `system_clock` on the other hand corresponds to the system-wide real-time clock, which can generally be set at will by the user. The `high_resolution_clock`, finally, is the clock with the shortest period supported by the library implementation (it may be an alias for `steady_clock` or `system_clock`).

To measure the time an operation took, a `steady_clock` should therefore be used, unless the `high_resolution_clock` of your implementation is steady:

```
using std::chrono::steady_clock;
const steady_clock::time_point before = steady_clock::now();
std::cout << steady_clock::period::num << '/'    /* Possible output: */
          << steady_clock::period::den << '\n';         // 1/1000000000
std::cout << (steady_clock::now() - before).count();    // 34721
```

The `system_clock` should be reserved for working with calendar time. Because the facilities of `<chrono>` in that respect are somewhat limited, this clock offers static functions to convert its `time_points` to `time_t` objects and vice versa (`to_time_t()` and `from_time_t()`, respectively), which can then be used with the C-style date and time utilities discussed in the next subsection:

```
using std::chrono::system_clock;
const auto now = system_clock::now();             /* Possible output: */
const time_t now_time_t = system_clock::to_time_t(now);
std::cout << now.time_since_epoch().count() << '\n';    // 1445470140000
std::cout << ctime(&now_time_t)<< '\n';      // Wed Oct 21 16:29:00 2015
```

C-Style Date and Time Utilities <ctime>

The <ctime> header defines two interchangeable types to represent a date and time:

- **time_t**: An alias for an arithmetic type (generally a 64-bit signed integer), represents time in a platform-specific manner.
- **tm**: A portable struct with these fields—tm_sec (range [0,60], where 60 is used for leap seconds), tm_min, tm_hour, tm_mday (day of the month, range [1,31]), tm_mon (range [0,11]), tm_year (year since 1900), tm_wday (range [0,6], with 0 being Sunday), tm_yday (range [0,365]), and tm_isdst (positive if Daylight Saving Time is in effect, 0 if not, and negative if unknown).

The following functions are available with <ctime>. The 'local time zone' is determined by the currently active C locale (locales are explained in Chapter 6):

Function	Returns
clock()	A clock_t (an arithmetic type) with the approximate processor time consumed by the process in clock ticks, or -1 upon failure. The clock's period is stored in the CLOCKS_PER_SEC constant. While this clock is steady, it may run at a different pace than wall clock time (slowed down due to context switches, sped up due to multithreading, etc.).
time()	Current point in time as a time_t, or -1 on failure. A time_t* argument must be passed: if not nullptr, the return value is written there as well.
difftime()	The difference between two time_ts as a double value denoting a time in seconds (result may be negative).
mktime()	A time_t, converted from a tm* for the local time zone, or -1 on failure.
localtime() gmtime()	A pointer to a statically allocated tm to which the conversion for the local/GMT time zone from a given time_t* has been written, or nullptr on failure. These functions are *not thread-safe*: this global tm is possibly shared among localtime(), gmtime(), and ctime().
asctime() ctime()	A char* pointer into a global buffer in which the (null-terminated) textual representation of a given tm* resp. time_t* is written, using a *fixed*, *locale-independent* format. As they are thus both limited and *not thread-safe*, they have been *deprecated* in favor of, e.g., strftime().
strftime()	Explained next.

Consult your implementation's documentation for safer alternatives for localtime() and gmtime() (e.g., localtime_s() for Windows or localtime_r() for Linux). For converting dates and times to strings, the preferred C-style function is strftime() (at the end of this section, we point out C++-style alternatives):

```
size_t strftime(char* result, size_t n, const char* format, const tm*);
```

An equivalent for converting to wide strings (`wchar_t` sequences), `wcsftime()`, is defined in `<cwchar>`. These functions write a `null`-terminated character sequence into `result`, which must point to a preallocated buffer of size `n`. If this buffer is too small, zero is returned. Otherwise, the return value shall equal the number of characters written, *not* including the terminating `null` character.

The grammar for specifying the desired textual representation is defined as follows: any character in the `format` string is copied to the `result`, except certain special specifiers which are replaced as shown in the following table:

Specifier	Output	Range or Example
%M / %S	Minutes / seconds	[00,59] / [00,60]
%H / %I	Hours using 24h / 12h clock	[00,23] / [01,12]
%R / %T	Equivalent to "%H:%M" / "%H:%M:%S"	04:29 / 04:29:00
%p / %r	A.m. or p.m. / full 12h clock time	pm / 04:29:00 pm
%A / %a	Full / abbreviated weekday name	Wednesday / Wed
%u / %w	Weekday number, where the first number in the range stands for Monday / Sunday	[1-7] / [0-6]
%d / %e	Day of the month	[01-31] / [1-31]
%j	Day of the year	[001-366]
%U / %V / %W	Week of the year, with weeks starting at Sunday (%U) or Monday (%V, %W); %V determines the first week of the year according to ISO 8601	[00,53] (%U,%W) / [01,53] (%V)
%B / %b, %h	Full / abbreviated month name (%h is an alias for %b)	October / Oct
%m	Month number	[01-12]
%Y / %G	Year / year current week belongs to per ISO 8601	2019
%C / %y / %g	First (%C) / last (%y, %g) two digits of the year. %g uses the year the current week belongs to per ISO 8601	20 / 19 / 19
%D / %F	Equivalent to "%m/%d/%y" / "%Y-%m-%d"	10/21/19 / 2019-10-21
%c / %x / %X	Preferred date + time / date / time representation	*(see later)*
%Z / %z	If available (empty if not): time zone name or abbreviation / offset from UTC as "±hhmm"	PDT / -0700
%% / %t / %n	Escaping / special characters	% / \t / \n

```
time_t time = std::time(nullptr);
tm time_tm = *std::localtime(&time);
char buffer[256];
strftime(buffer, sizeof(buffer), "Today is %a %e/%m%n", &time_tm);
std::cout << buffer;          // Today is Sat 30/01
strftime(buffer, sizeof(buffer), "%c--%x %X", &time_tm);
std::cout << buffer << '\n';  // Sat Jan 30 17:58:23 2016--01/30/16 17:58:23
```

The result of many specifiers, including those that expand to names or 'preferred' formats, depends on the active locale (cf. Chapter 6). When executed with a French locale, for example, the output for the preceding example could be "`Today is mer. 21/11`" and "`10/21/15 16:29:00--10/21/15 16:29:00`". To use a locale-dependent alternative representation (if one is defined by the current locale), `C`, `c`, `X`, `x`, `Y`, and `y` may be preceded by an `E` (i.e., `%EC`, `%Ec`, etc.); to use alternative numeric symbols, `d`, `e`, `H`, `I`, `M`, `m`, `S`, `u`, `U`, `V`, `W`, `w`, and `y` may be modified with the letter `O`.

As covered in Chapter 5, the C++ libraries offer facilities for reading/writing a `tm` from/to a stream as well, namely, `get_time()` and `put_time()`. The only C-style function from `<ctime>` you will generally need to output calendar dates and time in C++-style is therefore `localtime()` (to convert a `system_clock`'s `time_t` to `tm`).

Type Utilities

Runtime Type Identification `<typeinfo>, <typeindex>`

The C++ `typeid()` operator is used to obtain information on the runtime type of a value. It returns a reference to a global instance of the `std::type_info` class defined in `<typeinfo>`. These instances cannot be copied, but it is safe to use references or pointers to them. Comparison is possible using their `==`, `!=`, and `before()` members, and a `hash_code()` can be computed for them. Of particular interest is `name()`, which returns an implementation-specific textual representation of the value's type:

```
const std::string s = "Hello";
std::cout << typeid(s).name() << '\n';
std::cout << (typeid(typeid(s).name()) == typeid(s.data())); // 1 (true)
```

The `name()` printed may be something like "`std::basic_string<char,std::char_traits<char>,std::allocator<char>>`" (cf. Chapter 6), but for other implementations it might just as well simply be "`Ss`". The latter are so-called *mangled type names*, used by the compiler internally. Such names can generally be converted to a human-readable form using some compiler-specific function.

When used on a `B*` pointer to an instance of a derived class `D`, `typeid()` shall only give the dynamic type `D*` rather than the static type `B*` if `B` is polymorphic, that is, has at least one `virtual` member.

Because type_infos cannot be copied, they cannot be used as keys for the associative arrays of Chapter 3 directly. For precisely this purpose, the <typeindex> header defines the std::type_index decorator class: it mimics the interface of a wrapped type_info&, but it is copyable; has <, <=, >, and >= operators; and has a specialization of std::hash defined for it.

Type Traits **<type_traits>**

A *type-trait class,* or *type trait* for short, is a construct used to obtain compile-time information on a given type or to transform between related types. Type traits are used to inspect or manipulate template type arguments when writing generic code and are thus key components in *template metaprogramming*.

The <type_traits> header defines a multitude of trait classes. Due to page constraints, and because template metaprogramming is an advanced topic, this book cannot go into details on all of them. We will provide a brief reference though on the different type traits, which should be sufficient for basic usage.

Helper Classes

The <type_traits> header provides a std::integral_constant<T, T value> template that represents a constant of a given type. It serves as a base class for most type traits. The constant can be retrieved using the static value data member, by casting to T, or from its function call operator. Here is an example:

```
using answer_t = std::integral_constant<int, 42>;
std::cout << answer_t::value << std::endl; // static 'value' member
std::cout << answer_t{}<< std::endl;       // implicit casting operator
std::cout << answer_t{}()<< std::endl;     // function call operator
```

Additionally, the standard provides the following predefined constants:

- **std::true_type**, defined as integral_constant<bool, true>
- **std::false_type**, defined as integral_constant<bool, false>

C++17 adds a std::bool_constant<B> alias template defined as integral_constant<bool, B> C++17.

Type Classification

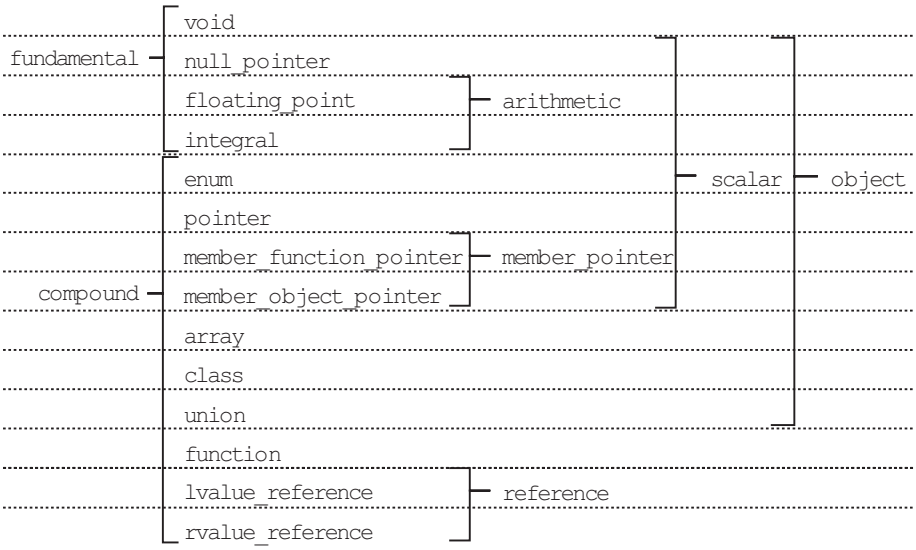


Figure 2-1. *Overview of the type classification traits. The second column lists the 14 primary categories; the other names are those of the composite categories.*

Each type in C++ belongs to exactly one of *14 primary type categories*. Besides those, the standard also defines several *composite type categories* to easily refer to all types belonging to two or more related primary categories. For each of these, a type trait `struct` exists to check whether a given type belongs to that category. Their names are of the form `is_`*category*, with *category* equal to one of the names shown in Figure 2-1. A trait's static Boolean named `value` contains whether its type argument belongs to the corresponding category. Traits are functors that both return and cast to this `value`.

Since C++17, all type traits that have a `value` member have corresponding helper variable templates defined to make them easier to use. These are called `std::`*trait*`_v<...>` and defined as `std::`*trait*`<...>::value`. For example, instead of writing `is_integral<int>::value`, you can simply write `is_integral_v<int>` C++17.

Some examples (the code refers to '`int main()`'):

```
std::cout << std::boolalpha;           // Print true/false instead of 1/0
std::cout << std::is_integral<int>::value << '\n';                // true
std::cout << std::is_integral_v<char> << '\n';                    // true
std::cout << std::is_class_v<std::is_class<bool>> << '\n';        // true
std::cout << std::is_function_v<int(void)> << '\n';               // true
std::cout << std::is_function_v<decltype(main)> << '\n';          // true
std::cout << std::is_pointer_v<decltype(&main)> << '\n';          // true
struct A { void f() {}; };
void(A::* p)() = &A::f;
std::cout << std::is_member_function_pointer<decltype(p)>() <<'\n';// true
```

Type Properties

A second series of type traits is there to statically query properties of types. They are mostly used in exactly the same manner as those of the previous subsection, and all except two, `has_virtual_destructor` and `has_unique_object_representations`, again have names of the form `is_`*property*.

The values for *property* listed as follows check the indicated type properties:

- The presence of type quantifiers: `const` and `volatile`
- Polymorphism properties of classes: `polymorphic` (has virtual member(s)), `abstract` (pure virtual member(s)), and `final`
- Signedness of arithmetic types: `signed` (includes floating-point numbers) and `unsigned` (includes Booleans)

And then there is a large family of traits where the property is the validity of a construction or assignment statement with specified argument types or the validity of a destruction statement (omitting as always the `'is_'`):

- The basic ones are `constructible<T,Args...>`, `assignable<T, Arg>`, and `destructible<T>`. All scalar types are destructible, and the former two properties may hold for nonclass types as well (as constructions like '`int i(0);`' for example, are valid).
- Auxiliary traits exist for checking the validity of default constructions (`default_constructible`) and copy/move constructions and assignments (`copy_constructible<T>` == `constructible<T, const T&>`, etc.).
- With the `swappable` and `swappable_with` traits, you can query whether objects of a certain type can be swapped or can be swapped with objects of a different type C++17.
- All preceding *property* names can mostly be prefixed with `trivially` or `nothrow`, for instance, `trivially_destructible`, `nothrow_constructible`, or `nothrow_swappable_with`. Only the `swappable(_with)` traits cannot be prefixed with `trivially`.

The `nothrow` properties hold if the corresponding operation is statically known to never throw. The `trivial` ones hold if the type is either `scalar` or a non-polymorphic class for which this operation is the default one (i.e., not specified by the user), and the `trivial` property holds as well for all its base classes and non-static member variables. For the trivially constructible properties, the class is also not allowed to have any non-static data members with in-class initializers.

The final list of *property* values for which `is_`*property* traits exist essentially holds under the following conditions. Arrays of types satisfying these also have the same property:

- **`trivially_copyable`**, if `trivially_destructible` and `trivially_(copy|move)_(constructible|assignable)` all hold. Bitwise copy functions such as `std::memcpy()` are defined to be safe for `trivially_copyable` types.

- **`trivial`**, if `trivially_default_constructible` and `trivially_copyable`, and no nondefault constructors exist.
- **`standard_layout`**, if `scalar` or a class for which a pointer to that class may safely be casted to a pointer to the type of its first non-static member (i.e., no polymorphism, limited multiple inheritance, etc.). This is for compatibility with C, as such casts (with C `structs` then) are common practice in C code.
- **`pod`** (plain old data), if `trivial` and `standard_layout`.
- **`aggregate`**, if `T` is an aggregate type, which is either an array type or a class type that has no private or protected data members, no user-provided and inherited constructors, no virtual methods, and no virtual, private, or protected base classes. With such class types (typically a `struct` or `union`), you can perform *aggregate initialization*; that is, you can initialize its members without having to write an explicit constructor C++17.
- **`literal_type`**, if values may be used in `constexpr` expressions (i.e., can be evaluated statically without side effects). This type property has been deprecated in C++17 and will be removed from C++20.
- **`empty`**, for non-polymorphic classes without non-static member variables.

Finally, the `std::has_unique_object_representations` trait determines whether `T` is a trivially copyable type for which two objects with the same value always have the exact same binary representation. You can use this to find out whether you may compute the hash of an object based on its representation as a sequence of bytes. An example for which this trait is typically `false` is `float`, as the IEEE-754 floating-point standard states that binary equality is not always the same as floating-point equality (for instance, `+0.f == -0.f`, but both values are represented using a different byte sequence). This trait will also be `false` whenever padding is added C++17.

Type Relationships

These three type traits compare types: `is_same<T1, T2>`, `is_base_of<Base, Derived>`, and `is_convertible<From, To>` (using implicit conversions).

Type Property Queries

The `value` constant of a type trait is not always a Boolean. For the following traits, it will contain the specified `size_t` type properties:

- **`std::alignment_of<T>`**: Value of `alignof(T)` operator
- **`std::rank<T>`**: Array dimensions, for example, `rank<int>() == 0`, `rank<int[]>() == 1`, `rank<int[][5][6]>() == 3`, and so on

- **`std::extent<T,N=0>`**: Number of elements of Nth array dimension, or 0 if unknown or invalid; for instance,

  `extent<int[]>() == 0` and `extent<int[][5][6],1>() == 5`

Type Transformations

Most type transformation traits are again fairly similar, except that they do not inherit from `std::integral_constant<>`. Instead of a static `value` member, they define a nested type alias called `type`. A convenience alias template with name `std::`*trait*`_t<T>` exists for all these traits, which is defined as `std::`*trait*`<T>::type`:

- **`std::add_x`** with **`x`** one of `const`, `volatile`, `cv` (`const` and `volatile`), `pointer`, `lvalue_reference`, `rvalue_reference`.
- **`std::remove_x`** with **`x`** one of `const`, `volatile`, `cv`, `pointer`, `reference` (lvalue or rvalue), `extent`, `all_extents`. In all except the last case, only the top-level/first type modifier is removed. For instance: `remove_extent<int[][5]>::type == int[5]`.
- **`std::decay<T>`** converts `T` to a related `type` that can be stored by value, mimicking by-value argument passing. An array type `int[5]`, for example, becomes a pointer type `int*`, a function a function pointer, `const`, and `volatile` are stripped, and so on. A possible implementation will be shown later as an example.
- **`std::make_signed`** and **`make_unsigned`**. If applied on an integral type `T`, `type` shall be a signed respectively unsigned integer type with `sizeof(type) == sizeof(T)`.
- **`std::underlying_type`**, defined only for `enum` types, gives the (integral) `type` underlying this `enum`.
- **`std::common_type<T...>`** has a `type` all types `T` can implicitly be converted to.
- **`std::conditional<B,T1,T2>`** has `type` `T1` if the `constexpr` `B` evaluates to `true`, and `type` `T2` otherwise.

The following example shows a possible implementation of the `std::decay` transformation trait in terms of the `std::conditional` metafunction. The latter is used to essentially form an `if-else` `if-else` construction at the level of types:

```
using namespace std;
template<typename T> struct my_decay {
private:
   using U = remove_reference_t<T>;
```

```
public:
   using type = conditional_t<is_array_v<U>, remove_extent_t<U>*,
           conditional_t<is_function_v<U>, add_pointer_t<U>,
           remove_cv_t<U>>>;
};
```

SFINAE Templates

SFINAE is an acronym for *Substitution Failure Is Not An Error* and is a rule in the C++ language that states that failure to specialize a template does not constitute a compile error. The following two templates are defined specifically to facilitate specific template metaprogramming patterns that exploit this principle:

- **`std::enable_if<B,T=void>`** has `type` `T`, but only if the `constexpr` `B` evaluates to `true`. Otherwise, `type` is not defined.
- **`std::void_t<...>`** maps any sequence of types to `void` C++17.

The examples in the upcoming subsections will clarify.

std::enable_if

Our first example shows how to leverage *SFINAE* to conditionally add or remove functions from overload resolution using `enable_if`. In this case, it shall be the absence of the type alias `type` that causes substitution to fail. For the first overload, we explicitly reference this alias (and are therefore forced to add `typename`); in the second, we instead use the more convenient template alias `enable_if_t`:

```
// use the efficient memcpy() if allowed (i.e., T is trivially copyable):
template<typename T, size_t N>
typename std::enable_if<std::is_trivially_copyable<T>::value>::type
copy(T(&from)[N], T(&to)[N])
{ std::memcpy(to, from, N * sizeof(T)); }

// otherwise, copy elements one by one using copy assignment:
template<typename T, size_t N>
std::enable_if_t<!std::is_trivially_copyable_v<T>>
copy(T(&from)[N], T(&to)[N])
{ for (size_t i = 0; i < N; ++i) to[i] = from[i]; }
```

With the introduction of `constexpr` `if` statements to the C++ language since C++17, most `std::enable_if` constructs can be greatly simplified. For example, instead of having to provide two overloads for the previous `copy()` function, the following implementation uses a single function which uses a `constexpr` `if` statement to select either the efficient or element-based copy algorithm at compile time C++17:

```
template<typename T, size_t N>
void copy(T(&from)[N], T(&to)[N])
{
  if constexpr (std::is_trivially_copyable_v<T>) {
    std::memcpy(to, from, N * sizeof(T));
  } else {
    for (size_t i = 0; i < N; ++i) to[i] = from[i];
  }
}
```

std::void_t C++17

The seemingly trivial utility trait `std::void_t` maps any sequence of types to `void`. You use it together with the SFINAE principle to verify the validity of expressions. For example, the following code defines a `has_value_type` `struct` to represent whether or not a given type `T` has a member called `value_type`:

```
template<typename, typename = std::void_t<>>
struct has_value_type : std::false_type {};

template<typename T>
struct has_value_type<T, std::void_t<typename T::value_type>>
  : std::true_type {};

template<typename T>
inline constexpr bool has_value_type_v = has_value_type<T>::value;

// All containers (Chapter 3) define a value_type type alias (example
// needs the <vector> header), as does std::integral_constant itself:
static_assert(has_value_type<std::vector<int>>::value);
static_assert(!has_value_type_v<int>);
static_assert(has_value_type<has_value_type<int>>{});
```

Function Invocation Traits C++17

C++17 adds some more traits related to properties of function invocations:

- **`is_invocable<F,Args...>`** / **`is_nothrow_invocable<F,Args...>`** determine whether the callable `F` can be called with the given argument types (for the latter: without throwing exceptions).
- **`is_invocable_r<R,F,A...>`** / **`is_nothrow_invocable_r<R,F,A...>`** are analogous to the `is_invocable` traits, but additionally check whether this invocation results in a type that is convertible to `R`.
- **`invoke_result<F,Args...>`** has a `type` member containing the return type of a given callable when called with the given argument types.

This last trait replaces `std::result_of<F(Args...)>` which has been deprecated in C++17 and will be removed from C++20.

Trait Operations C++17

Zero or more type traits that have a `value` member that is convertible to a Boolean (such as all type 'is_' and 'has_' traits of the `<type_traits>` header) can be combined using these logical operations:

- **`std::conjunction`** performs a logical AND operation on traits.
- **`std::disjunction`** performs a logical OR operation on traits.
- **`std::negation`** performs a logical NOT operation on a trait.

Both `conjunction` and `disjunction` support the equivalent of short-circuiting of Boolean expressions with respect to SFINAE. That is, conjuncts and disjuncts are instantiated left to right, and instantiation stops once the correct `value` has been determined, even if any of the remaining conjuncts or disjuncts would have led to a substitution failure.

Type Operations <utility>

std::declval

The return type of `std::declval<T>()` is `T&&`; only when `T` is `void`, the return type is `T`. This function has no implementation, so evaluating it is an error. It can only be used in so-called *unevaluated expressions*, which are expressions that are passed to `decltype()`, `noexcept()`, `typeid()`, or `sizeof()`. Such expressions are interpreted at compile time, never evaluated.

You typically use `declval<T>()` to invoke a member function on an object of type `T` in an unevaluated expression without having to know which constructors exist for `T`, often in combination with type traits. As the following example shows, `declval<T>()` even works if `T` has no default constructor (this unlike `T{}`):

```
template<typename T>
void process(const T& /*t*/) {
  if constexpr (std::is_integral_v<decltype(std::declval<T>().handle())>)
     std::cout << "handle() has integral return type\n";
  else
    std::cout << "handle() has non-integral return type\n";
}

struct NoDefaultCtor { NoDefaultCtor(int){}; int handle(); };
struct DefaultCtor{ std::string handle(); };

int main() {
  process(NoDefaultCtor{0}); // handle() has integral return type
  process(DefaultCtor{});    // handle() has non-integral return type
}
```

std::as_const C++17

Given a reference, `std::as_const()` returns a `const` reference. An overload accepting an rvalue reference is deleted to prevent calling `as_const()` with rvalue references.

This function comes in handy, for instance, when applying Scott Meyer's guideline for avoiding code duplication with class methods that are overloaded on `const`ness. It encourages you to implement `const` versions of class methods as you normally would and to implement non-`const` versions by forwarding to the `const` versions. We also refer to this pattern as the *to-const-and-back-again* pattern. Here is a quick example:

```
class Matrix
{
 public:
  struct Cell { /* ... */};
  Cell& at(size_t x, size_t y);
  const Cell& at(size_t x, size_t y) const;
  /* ... */
};

const Matrix::Cell& Matrix::at(size_t x, size_t y) const {
  // Some otherwise duplicated logic: input verification, ...
  return /* ... */;
}
Matrix::Cell& Matrix::at(size_t x, size_t y) {
  // Before: const_cast<Cell&>(static_cast<const Matrix&>(*this).at(x, y));
  return const_cast<Cell&>(std::as_const(*this).at(x, y));
}
```

Generic Utilities

The functions in this section generalize a common operation (function invocation and taking the address of an object, respectively) such that these work correctly even when applied on corner cases that would otherwise require special syntax (member function pointers and types with an overloaded `operator&()`, respectively). As such, these functions are particularly useful as well when writing function templates.

std::invoke C++17 **<functional>**

`std::invoke()` invokes a given callable with a given set of arguments. The callable can be a free function, a lambda closure, a functor, a method on an object, and so on. The advantage of `std::invoke(f, args...)` compared to simply invoking `f(args...)` is that

it also works if f is a member function pointer, which normally needs a special syntax. This is shown in the following example:

```
template<typename C, typename... Args>
void process(C callable, Args... args) {
  // ... Do stuff ...
  std::invoke(callable, std::forward<Args>(args)...);
  // ... Do more stuff ...
}

class Processor {
 public: void handle(int data) { std::cout << "Processor::handle()\n"; }
};
void f(float data) { std::cout << "Global function f()\n"; }

int main() {
  Processor processor;
  process(&f, 15.85714f);
  process(&Processor::handle, processor, 42);
}
```

std::addressof <memory>

Usually, the address of an object can be retrieved using `operator&`. However, objects are allowed to overload this operator to give it a different behavior. Although this is rarely done, if an object does overload `operator&`, that operator cannot be used (directly) anymore to get the real address of the object. In such cases, you can use `std::addressof()` to get the true address in memory. You should also use `addressof()` to retrieve the address of an object of unknown type in generic code.

CHAPTER 3

Containers

The C++ Standard Library provides a selection of different data structures called *containers* that you can use to store collections of data elements. Containers work in tandem with *algorithms*, described in Chapter 4. Containers and algorithms are designed in such a way that they do not need to know about each other. The interaction between them is accomplished with *iterators*. All containers provide iterators, and algorithms only need iterators to be able to perform their work.

This chapter starts by explaining the concept of iterators followed by a description of all sequential containers, container adaptors, and associative containers. We end the chapter with a brief description of C++ *allocators*.

Iterators

<iterator>

Iterators are the glue between containers and algorithms. They provide a way to enumerate all elements of a container in a uniform way without having to know any details about the container. The following list briefly mentions the most important iterator categories provided by the standard, and the subsequent table explains all the operations possible on them:

- *Forward (F)*: An input iterator that supports forward iteration
- *Bidirectional (B)*: A forward iterator that can move forward and backward
- *Random access (R)*: A bidirectional iterator that support jumping to elements at arbitrary indices or offsets

P. Van Weert and M. Gregoire, *C++17 Standard Library Quick Reference*,
https://doi.org/10.1007/978-1-4842-4923-9_3

In the following table, `T` is an iterator type, `a` and `b` are instances of `T`, `t` is an instance of the type to which `T` points, and `n` is an integer.

Operation	Description	F	B	R
`T a`, `~T()`, `T b(a)`, `b = a`	Default constructor, destructor, copy constructor, copy assignment	■	■	■
`a == b`, `a != b`	Equality and inequality operators	■	■	■
`*a`, `a->m`, `*a = t`, `*a++ = t`	Dereferencing	■	■	■
`++a`, `a++`, `*a++`	Incrementing operators	■	■	■
`--a`, `a--`, `*a--`	Decrementing operators	□	■	■
`a[n]`	Random access	□	□	■
`a + n`, `n + a`, `a - n`, `a += n`, `a -= n`	Arithmetic operators. Advance an iterator forward or backward	□	□	■
`a - b`	Calculates the distance between iterators	□	□	■
`a < b`, `a > b`,`a <= b`, `a >= b`	Inequality operators	□	□	■

From this, it is obvious that random iterators are very similar to C++ pointers. In fact, pointers into a regular C-style array satisfy all requirements for a random access iterator and can therefore be used with the algorithms from Chapter 4 as well. Certain sequential containers likely even use regular pointers for iterators (through type aliasing). For more complex data structures, though, this is not possible, and iterators are implemented as small classes.

All Standard Library compliant containers must provide an `iterator` and `const_iterator` member type. Additionally, containers that support reverse iteration must provide the `reverse_iterator` and `const_reverse_iterator` member types. For example, the reverse iterator type for a `vector` of integers is `std::vector<int>::reverse_iterator`.

Iterator Tags

The type trait expression `std::iterator_traits<Iter>::iterator_category` may be used by generic algorithms to optimize their implementation based on the category of its iterator arguments (typically in combination with, for instance, `std::enable_if` or C++17's `constexpr if` statements). The `std::distance()` function explained in an upcoming section, for instance, may use this technique to choose between a slower implementation that linearly calculates the distance between two iterators (for forward and bidirectional iterators) and a more efficient one that simply subtracts two iterators (for random access iterators).

For a given `Iter` type, `iterator_category` evaluates to a type alias for one of the so-called *iterator tags*, empty types that are defined solely for their use in template

metaprogramming. These tag types are named `std::`*category*`_iterator_tag`, where possible *category* values include `forward`, `bidirectional`, and `random_access`.[1]

If you implement your own iterators, you should therefore specify its tag. You can do this either by adding a static type alias member `iterator_category` to your implementation that refers to one of the iterator tag types or by specializing `std::iterator_traits` for your type to provide the correct tag type.

■ **Note** The `<iterator>` header also defines a `std::iterator<>` class template that was intended to serve as the base class of custom iterator types. It defines, among other things, the `iterator_category` alias. This template is deprecated, however. C++17

Non-member Functions to Get Iterators

All containers support member functions that return various iterators. However, the standard also provides non-member functions that can be used to get such iterators. In addition, these non-member functions work the same way on containers, C-style arrays, and `initializer_lists`. The provided non-member functions are as follows:

Non-member Function	Description
`begin()` / `end()`	Returns an iterator to the first, or, respectively, one past the last element
`cbegin()` / `cend()`	`const` versions of `begin()` and `end()`
`rbegin()` / `rend()`	Returns a reverse iterator to the last, or, respectively, one before the first element
`crbegin()` / `crend()`	`const` versions of `rbegin()` and `rend()`

Dereferencing the iterators returned by the `const` versions, also called *const iterators*, results in `const` references and therefore cannot be used to modify the elements in the container or array. A *reverse iterator* allows you to traverse a container's elements in reverse order: starting with the last element and going toward the first element. When you increment a reverse iterator, it actually moves to the previous element in the underlying container.

Here is an example of how to use such non-member functions on a C-style array:

```
int myArray[] = { 1,2,3,4 };
const auto beginIter = std::cbegin(myArray);
const auto endIter = std::cend(myArray);
for (auto iter = beginIter; iter != endIter; ++iter)
   std::cout << *iter << std::endl;
```

[1] Two more iterator categories and corresponding tag types are introduced in the next chapter. These are not relevant in the context of containers though.

However, instead of this, it is recommended that you use a range-based `for` loop to iterate over all elements of a C-style array or Standard Library container. It is much shorter and clearer. For example:

```
int myArray[] = { 1,2,3,4 };
for (const auto& element : myArray)
   std::cout << element << std::endl;
```

You cannot always use the range-based `for` loop version, though. If you want to loop over the elements and remove some of them, for instance, then you need the iterator version.

Non-member Operations on Iterators

The following non-member operations exist to perform random access operations on all types of iterators. When called on iterators that are not known to support random access (see earlier), the implementation automatically falls back to a method that works for that iterator (e.g., a linear traversal):

- `std::distance(iter1, iter2)`: Returns the distance between two iterators.
- `std::advance(iter, dist)`: Advances an iterator by a given distance and returns nothing. The distance can be negative if the iterator is bidirectional or random access.
- `std::next(iter, dist=1)`: Equivalent to `advance(iter, dist)` and returns `iter`.
- `std::prev(iter, dist=1)`: Equivalent to `advance(iter, -dist)` and returns `iter`. Only works for bidirectional and random access iterators.

Sequential Containers

The following sections describe the five sequential containers: `vector`, `deque`, `array`, `list`, and `forward_list`. The `std::vector` container is discussed in more detail compared to the others. Once you know how to work with one container, however, you know how to work with others. At the end of this section is a reference with all available methods supported by sequential containers.

std::vector **<vector>**

A `vector` stores its elements contiguously in memory. It is comparable to a heap-allocated C-style array, except that it is safer and easier to use because `vector` automatically releases its memory and grows to accommodate new elements.

Construction

Like all Standard Library containers, `vector` is templated on the type of object stored in it. The following piece of code shows how to define a `vector` of integers:

```
std::vector<int> myVector;
```

Initial elements can be specified using a braced initializer:

```
std::vector<int> myVector1 = { 1,2,3,4 };
std::vector<int> myVector2{ 1,2,3,4 };
```

You can also construct a `vector` with a certain size. For example:

```
std::vector<int> myVector(100, 1);
```

This creates `myVector` containing `100` elements with value `1`. The second argument is optional. If you omit it, new elements are zero-initialized, which is `0` for the case of integers.

■ **Caution** Take care with uniform initialization when constructing `vectors` of numbers. The following statement, for instance, does not construct a `vector` containing 100 values equal to 1, but a `vector` containing just 2 values—100 and 1:

```
std::vector<int> myVector{100, 1};
```

Starting with C++17, you can omit template type arguments when you construct a container if these types can be deduced from the constructor arguments C++17:

```
std::vector oneTwoThree{ 1,2,3 }; // <int> deduced from initializer list
std::vector fiveFloats(5, 0.f);   // <float> deduced from 0.f
```

Accessing Elements

Elements in a `vector` can be accessed using `operator[]`, which returns a reference to an element at a specific zero-based index, making it behave exactly as with C-style arrays. For example:

```
myVector[1] = 22;
std::cout << myVector[1];  // 22
```

No bounds checking is performed when using `operator[]`. If you need bounds checking, use the `at()` method: it throws a `std::out_of_range` exception if the given index is out of bounds.

`front()` can be used to get a reference to the first element, and `back()` returns a reference to the last element.

To access the contiguous array of dynamic memory holding a `vector`'s elements, you use the `data()` method. This is particularly useful when interfacing with functions that expect a C-style array:

```
int* data = myVector.data();
```

■ **Note** In older code, you often see `&myVector[0]` instead, but this expression is inherently unsafe for empty containers. Even with `data()` you need to be careful though: it is implementation-defined whether `data()` may or may not return `nullptr` for an empty `vector` (rather than a pointer to an empty, null-terminated array).

Iterators

`vector` supports random access iterators. You use the `begin()` or `cbegin()` member to get a non-`const` or `const` iterator to the first element in the `vector`. The `end()` and `cend()` methods are used to get an iterator to one past the last element. `rbegin()` and `crbegin()` return a reverse iterator to the last element, and `rend()` and `crend()` return a reverse iterator to one before the first element.

As always, you can also use the equivalent non-member functions explained earlier, such as `std::begin()`, `std::cbegin()`, and so on.

Adding Elements

One way to add elements to a `vector` is to use `push_back()`. For example, adding two integers to `myVector` can be done as follows:

```
std::vector<int> myVector;
myVector.push_back(11);
myVector.push_back(2);
```

Another option is to use the `insert()` method, which requires an iterator to the position before which the new element should be inserted. For example:

```
std::vector myVector{ 1,2,3,4 };
myVector.insert(myVector.begin() + 2, 22); // 1,2,22,3,4
```

Just like any modifying operation, insertion generally invalidates existing iterators. So when inserting in a loop, the following idiom should be used:

```
std::vector myVector{ 1,2,3,4 };
for (auto iter = myVector.begin(); iter != myVector.end(); ++iter)
{
 if (*iter % 2 == 0)       // Duplicate all even values...
   iter = myVector.insert(iter + 1, *iter);
}
```

This works because `insert()` returns a valid iterator pointing to the inserted element (more generally, to the first inserted element, discussed shortly). Also make sure never to cache the end iterator in such loops, as `insert()` might invalidate it.

With `insert()` you can also insert a range of elements anywhere in the `vector`, or concatenate (append) two vectors. When doing so, you do not have to resize the `vector` yourself; `insert()` takes care of that for you. For example:

```
std::vector v1{ 1,2,3 };
std::vector v2{ 4,5 };
v1.insert(cbegin(v1)+1, cbegin(v2), cend(v2)); // 1,4,5,2,3
v1.insert(cend(v1), cbegin(v2), cend(v2));     // 1,4,5,2,3,4,5 (append!)
```

Two additional overloads of `insert()` provide insertion of initializer lists or a given number of copies of a certain element. Using the same initial `v1` as before:

```
v1.insert(cbegin(v1)+1, {4,5});       // 1,4,5,2,3
v1.insert(cend(v1), 2, 6);            // 1,4,5,2,3,6,6
```

Instead of constructing a new element and then passing it to `insert()` or `push_back()`, elements can also be constructed in place using an *emplacement* method, such as `emplace()` or `emplace_back()`. The former, `emplace()`, is the counterpart of a single-element `insert()`, the latter of `push_back()`. For example, you can add a new `Person` at the back of a `vector` of `Person` objects in these two similar ways:

```
std::vector<Person> persons;
persons.push_back(Person("Sheldon", "Cooper"));
persons.emplace_back("Leonard", "Hofstadter");
```

The arguments to emplacement functions are perfect-forwarded to the element's constructors. Emplacement is generally more efficient if it avoids the creation of a temporary object, as in the previous example. This is particularly interesting if copying is expensive. It may even be the only way to add elements that cannot be copied.

On a related note: addition and insertion members of containers generally have full support for moving elements into containers, again to avoid the creation of unnecessary copies (move semantics is explained in Chapter 2). For example:

```
Person person("Howard", "Wolowitz");
persons.push_back(std::move(person));
```

Size and Capacity

A `vector` has a size, which is the number of elements contained in the `vector`. You can query it using `size()`. Use `empty()` to check whether a `vector` is empty or not.

A `vector` can be resized with `resize()`. For example:

```
std::vector<int> myVector;
myVector.resize(100, 12);
```

This sets the size of the `vector` to 100 elements. If new elements have to be created, they are initialized with 12. The second parameter is again optional; when omitted, new elements are zero-initialized.

In addition to a size, a `vector` also has a capacity, returned by `capacity()`. The capacity is the total number of elements it can store (including the elements already in the `vector`) without having to allocate more memory. If more elements are added than allowed by the capacity, the `vector` must perform a reallocation because it needs to store all elements contiguously in memory. Reallocation means that a new, bigger block of memory is allocated and that all current elements in the `vector` are transferred to the new location (they are moved if moving is supported and known not to throw; otherwise they are copied; see Chapter 2).

If you know how many elements you will add, it often benefits performance to preallocate sufficient capacity to avoid reallocation. Failure to do so may cause a significant performance hit. You can preallocate memory using `reserve()`:

```
myVector.reserve(100);
```

Note that this does not reserve capacity for 100 *extra* elements; it simply ensures that the total capacity of `myVector` is at least 100. Reserving capacity for a nonempty `vector` to store 100 extra elements should be done as follows:

```
myVector.reserve(myVector.size() + 100);
```

Removing Elements

The last element in a `vector` can be removed using `pop_back()`, and `erase()` is used to remove other elements. There are two overloads of `erase()`:

- `erase(iter)`: Removes the element to which the given iterator points
- `erase(first, last)`: Removes the range of elements given by the two iterators, so [`first`, `last`)

When you remove elements, the size of the `vector` changes, but its capacity does not. If you want to reclaim unused memory, you can use `shrink_to_fit()`. This is just a hint, however, which may be ignored by an implementation.

To remove all elements, you can use `clear()`. This again does not affect capacity. A classic idiom to clear a container while guaranteeing its memory is reclaimed is to swap with an empty one:

```
std::vector unlucky(100000, 13);   // Now reclaim unlucky's memory...
{ std::vector<int> empty; empty.swap(unlucky); }
```

When the scope exits, the formerly empty container is destroyed containing all elements, leaving the original one empty. This idiom may be written more briefly as follows:

```
std::vector<int>().swap(unlucky);  // (temporary is destroyed after ';')
```

■ **Caution** Do not mistake `empty()` for `clear()` when you intend to remove all elements of a container. `empty()` simply returns a Boolean and does not empty the container at all.

Remove-Erase Idiom

If you need to remove a number of elements from a `vector`, you can write your own loop to iterate over all the elements, and invoke `erase()` for all elements that need to be removed. The problem, however, is that `erase()` invalidates all iterators pointing at or beyond the element that was removed. As this includes both the current iterator and the `end()` iterator, you have to be extra careful not to make any mistakes.

The following example shows the typical pattern that you could use to safely remove all elements equal to 2 from a vector:

```
std::vector vec{ 1,2,3,2,2,6 };
for (auto it = cbegin(vec); it != cend(vec);) {
  if (*it == 2)
     it = vec.erase(it);  // Returns iterator one past the removed item
  else
     ++it;
}
```

Notice that the old `it` is no longer used after `erase()` and that the end iterator must certainly not be cached either.

To avoid mistakes, it is always recommended that you use standard algorithms instead of hand-written loops. To remove multiple elements, you can use the *remove-erase idiom*. This pattern first uses the `std::remove()` or `remove_if()` algorithm. As we explain in Chapter 4, these algorithms do not actually remove elements. Instead, they move all elements that need to be kept toward the beginning, maintaining the relative order of these elements. The algorithms return an iterator to one past the last element to be kept.

The next step usually is to call `erase()` on the container to really erase the elements starting from the iterator returned by `remove()` or `remove_if()` to the end. For example:

```
std::vector vec{ 1,2,3,2,2,6 };                    // 1,2,3,2,2,6
auto iter = std::remove(begin(vec), end(vec), 2); // 1,3,6,2,2,6
vec.erase(iter, end(vec));                         // 1,3,6
```

The call to `remove()` in the second line moves all elements to keep toward the beginning of the `vector`. The contents of the other elements (i.e., those to remove) can be different depending on your library implementation.

The previous `remove()` and `erase()` calls can also be combined into one line:

```
vec.erase(std::remove(begin(vec), end(vec), 2), end(vec));
```

■ **Caution** A common mistake (that can go unnoticed for quite a while) when using the remove-erase idiom is forgetting to pass the end iterator as the second argument to `erase()`. This argument is marked in bold in our examples. Without it, you only delete one element:

```
vec.erase(std::remove(begin(vec), end(vec), 2));  // 1,3,6,2,6
```

---

std::vector<bool>

`vector<bool>` is a specialization of `vector<T>` for Boolean elements. It allows C++ Standard Library implementations to store the Boolean values in a space-efficient way, but this is not a requirement. It has the same interface as `vector<T>`, with the addition of a `flip()` method to flip all the bits in the `vector<bool>`.

This specialization is similar to the `std::bitset` discussed later. The difference is that a `bitset` has a fixed size, whereas a `vector<bool>` can dynamically grow and shrink as needed.

Both `vector<bool>` and `bitset` are recommended only to save memory; otherwise, use a `vector<std::uint_fast8_t>`: this generally has superior performance when it comes to accessing, traversing, or assigning values.

Complexity

The complexity of common operations on a `vector` is as follows:

- *Insertion*: Amortized constant O(1) at the end; otherwise linear in the distance from the insertion point to the end of the vector, O(N)
- *Deletion*: O(1) at the end, otherwise linear in the distance to the end of the vector O(N)
- *Access*: O(1)

Even though `list` and `forward_list`, discussed later, have better theoretical insertion and deletion complexities, `vector` is typically faster in practice and should therefore be your default sequential container. When in doubt, always use a profiler to compare their performance for your application.

std::deque <deque>

Much like a `vector`, a `deque`[2] is a sequential container that supports random access to its elements, both through `operator[]` and its iterators. Unlike a `vector`, however, a `deque` does not store all its elements in one contiguous array. Instead, it arranges its data such that insertions and removals at *both* ends of the queue—*deque* is short for *double-ended queue*—have the following appealing properties:

1. They always occur in constant time.
2. They never require any other elements to be copied or moved.
3. They never invalidate pointers or references to the other elements.

For a `vector`, these properties only hold when altering the back of the array, and then only if you do so without exceeding the capacity.

The operations on a `deque` are almost the same as for a `vector`, with a few minor differences. A `deque` also does not have the concept of capacity, so `capacity()`, `reserve()`, and `shrink_to_fit()` are not available. Moreover, a `deque` provides (constant time) `push_front()` and `pop_front()` methods in addition to `push_back()` and `pop_back()`.

Here is an example of using a `deque`:

```
std::deque myDeque{ 1,2,3,4 };           // std::deque<int> deduced
myDeque.insert(myDeque.begin() + 2, 22); // 1,2,22,3,4
myDeque.pop_front();                     // 2,22,3,4
myDeque.erase(myDeque.begin() + 1);      // 2,3,4
myDeque.push_front(11);                  // 11,2,3,4
```

Complexity

The complexity of common operations on a `deque` is as follows:

- *Insertion*: O(1), constant, at both ends of the queue (front and back); otherwise O(N), linear, in the distance between the insertion point and the beginning or end
- *Deletion*: O(1) at the front and back; otherwise O(N), linear, in the distance to the beginning or end
- *Access*: O(1)

[2] Deque is pronounced /dɛk/, like deck (as in a deck of cards).

std::array <array>

An `array` is a container with a fixed size that is specified at compile time as a template argument. It again supports `operator[]` and random access iterators.

The following defines an array of three integers:

```
std::array<int, 3> myArray;
```

These integers are *uninitialized*. All other containers zero-initialize their elements by default. This is because a `std::array` is designed to be as close as possible to a C array. Of course, you can also initialize elements when defining an `array`. The number of initialization values must equal the size of the `array` or less. If you specify more values, you get a compilation error. Elements for which no value is specified are zero-initialized. For example:

```
std::array<int, 3> myArray{ 1,2 };  // 1,2,0
```

This also implies that the following zero-initializes all elements:

```
std::array<int, 3> myArray{};  // 0,0,0
```

The template arguments can be deduced from a (nonempty) initializer list:

```
std::array myArray{ 1,2 };     // std::array<int, 2> deduced
```

There is one special method, `fill()`, which fills the `array` with a certain value. For example:

```
myArray.fill(5);  // 5,5,5
```

Complexity

- *Insertion*: Not possible
- *Deletion*: Not possible
- *Access*: O(1)

std::list and std::forward_list <list>, <forward_list>

A `list` stores its elements as a doubly linked list, whereas a `forward_list` stores them as a singly linked list. This has some consequences.

Firstly, because both store elements noncontiguously in memory, you cannot access arbitrary elements of these containers in constant time. For this reason, `list` and `forward_list` also do not provide `operator[]`. To access a specific element of either list container, you always have to perform a linear search using iterators. `list` supports bidirectional iterators, so you can start at the beginning or the end; `forward_list` only supports forward iterators, so you always need to start at the beginning.

Once you are at the correct place in the container, however, insertions and deletions at that place are efficient because they only need to modify a couple of links. These insertions and deletions never invalidate iterators or references to other elements either.

Next to the lack of random access, a second downside is that the elements of list-based containers may become scattered in memory, which hurts performance when traversing lists due to an increased number of cache misses.

■ **Tip** Because of their downsides, only use a `list` or `forward_list` if a profiler shows that it is more efficient for your use case. Mostly a `vector` will be more efficient, however, even if you need frequent insertions or removals.

The operations supported by `list` and `forward_list` are again similar to those of a `vector`, with some minor differences. A `list` or `forward_list` does not have a capacity, so none of the capacity-related methods are supported. Both support `front()`, which returns a reference to the first element. A `list` also supports `back()` returning a reference to the last element. `forward_list` is the only container without a `size()` member.

Complexity

`list` and `forward_list` have similar complexities:

- *Insertion*: O(1) once you are at the correct position
- *Deletion*: O(1) once you are at the correct position
- *Access*: O(1) to access the first (for `list` and `forward_list`) or last (only for `list`) element; otherwise O(N)

List-Specific Algorithms

Due to the nature of how `list` and `forward_list` store their elements, they provide a couple of member functions that implement specific algorithms. The following table lists the provided algorithms for `list` (L) and `forward_list` (F):

Operation	L	F	Description
`merge()`	■	■	Merges two sorted lists. The list that is merged in is emptied.
`remove()`	■	■	Removes elements from the list that match a given value.
`remove_if()`	■	■	Removes elements from the list that satisfy a given predicate.
`reverse()`	■	■	Reverses the contents of the list.
`sort()`	■	■	Sorts the elements.
`splice()`	■	□	Moves elements from another list *before* a given position.
`splice_after()`	□	■	Moves elements from another list *after* a given position.
`unique()`	■	■	Replaces consecutive duplicates with a single element.

For all of these algorithms except `splice()` and `splice_after()`, generic versions are available that are explained in Chapter 4. These generic versions work on all types of containers, but the list containers provide special implementations that are more efficient.

Here is an example of using some of these list algorithms:

```
std::list list1{ 1,7,5 }, list2{ 5,6,2 }, list3{ 3,4 }; // list<int>
list1.sort();        // 1,5,7
list2.sort();        // 2,5,6
list1.merge(list2);  // list1 = 1,2,5,5,6,7
                     // list2 = empty
list1.unique();      // 1,2,5,6,7

auto splicePosition = std::next(begin(list1), 2);
list1.splice(splicePosition, list3); // list1 = 1,2,3,4,5,6,7
                                     // list3 = empty
```

Sequential Containers Reference

The following subsections give an overview of all the operations supported by `vector` (V), `deque` (D), `array` (A), `list` (L), and `forward_list` (F), divided into categories.

Iterators

Operation	V	D	A	L	F	Description
`begin()` `end()`	■	■	■	■	■	Returns an iterator to the first or one past the last element
`cbegin()` `cend()`	■	■	■	■	■	`const` versions of `begin()` and `end()`
`rbegin()` `rend()`	■	■	■	■	□	Returns a reverse iterator to the last element or one before the first element
`crbegin()` `crend()`	■	■	■	■	□	`const` versions of `rbegin()` and `rend()`
`before_begin()`	□	□	□	□	■	Returns an iterator to the element right before the element returned by `begin()`
`cbefore_begin()`	□	□	□	□	■	`const` version of `before_begin()`

Size and Capacity

Operation	V	D	A	L	F	Description
`size()`	■	■	■	■	□	Returns the number of elements
`max_size()`	■	■	■	■	■	Returns the maximum number of elements that can be stored in the container
`resize()`	■	■	□	■	■	Resizes the container
`empty()`	■	■	■	■	■	Returns `true` if the container is empty, `false` otherwise
`capacity()`	■	□	□	□	□	Returns the current capacity of the container
`reserve()`	■	□	□	□	□	Reserves capacity
`shrink_to_fit()`	■	■	□	□	□	Hint to reduce the capacity of the container to match its size

Access

Operation	V	D	A	L	F	Description
`operator[]`	■	■	■	□	□	Returns a reference to an element at a given index position. No bounds checking is performed on the index.
`at()`	■	■	■	□	□	Returns a reference to an element at a given index position. If the given index position is out of bounds, a `std::out_of_range` exception is thrown.
`data()`	■	□	■	□	□	Returns a pointer to the data of the `vector` or `array`. May return `nullptr` if `size()` equals zero.
`front()`	■	■	■	■	■	Returns a reference to the first element. Undefined behavior on an empty container.
`back()`	■	■	■	■	□	Returns a reference to the last element. Undefined behavior on an empty container.

Modifiers

Operation	V	D	A	L	F	Description
`assign()`	■	■	■	■	■	Replaces the contents of the container with • N copies of a given value, or • Copies of elements from a given range, or • Elements from a given `initializer_list`
`clear()`	■	■	□	■	■	Deletes all elements; size becomes zero.
`emplace()`	■	■	□	■	□	Constructs a single new element in place before the element pointed to by a given iterator. The iterator argument is followed by zero or more arguments that are just forwarded to the element's constructor.
`emplace_back()`	■	■	□	■	□	Constructs a single new element in place at the end.
`emplace_after()`	□	□	□	□	■	Constructs a single new element in place after an existing element.
`emplace_front()`	□	■	□	■	■	Constructs a single new element in place at the beginning.
`erase()`	■	■	□	■	□	Erases elements.
`erase_after()`	□	□	□	□	■	Erases an element after an existing iterator position.
`fill()`	□	□	■	□	□	Fills the container with a given element.
`insert()`	■	■	□	■	□	Inserts one or more elements before the element pointed to by a given iterator.
`insert_after()`	□	□	□	□	■	Inserts one or more elements after the element pointed to by a given iterator.
`push_back()` `pop_back()`	■	■	□	■	□	Adds an element at the end, or, respectively, removes the last element.
`push_front()` `pop_front()`	□	■	□	■	■	Adds an element at the beginning, or, respectively, removes the first element.
`swap()`	■	■	■	■	■	Swaps the contents of two containers in constant time, except for `array`s, where it needs linear time.

Non-member Functions

Sequential containers support the following non-member functions:

Operation	Description
==, !=, <, <=, >, >=	Compares values in two containers (lexicographically)
`std::swap()`	Swaps the contents of two containers
`std::size()`	Returns the number of elements (not for `std::forward_list`)
`std::empty()`	Returns whether the container is empty
`std::data()`	Same as the `data()` member function (`array` and `vector` only)

`std::swap()`, `size()`, `empty()`, and `data()` all work for non-container types as well—and C-style arrays in particular—which makes them particularly useful in function templates that operate on arbitrary data ranges. Outside the context of templates, you'll probably find `std::size()` the most useful function to remember.[3] Here is an example:

```
int array[] { 1, 2, 3};
std::cout << std::size(array) << std::endl;   // 3
```

This is clearly far more convenient than the old-school alternative:

```
std::cout << sizeof(array) / sizeof(array[0]) << std::endl;
```

The <array> header defines one additional non-member function, `std::get<Index>()`, and helper types `std::tuple_size` and `std::tuple_element`, which are equivalent to the same function and types defined for tuples and pairs explained in Chapter 2.

std::bitset <bitset>

A `bitset` is a container storing a fixed number of bits. The number of bits is specified as a template parameter. For example, the following creates a `bitset` with 10 bits, all initialized to 0:

```
std::bitset<10> myBitset;
```

The values for the individual bits can be initialized by passing an integer to the constructor or by passing in a string representation of the bits. For example:

```
std::bitset<4> myBitset("1001");
```

[3] `std::size()`, `data()`, and `empty()` are defined by various Standard Library headers, among which <iterator> and all container headers. We find it easiest to include the <array> header though when using, for instance, `std::size()` on C-style arrays.

A `bitset` can be converted to an integer or a string with `to_ulong()`, `to_ullong()`, and `to_string()`.

Complexity

- *Insertion*: Not possible
- *Deletion*: Not possible
- *Access*: O(1)

Reference

Access

Operation	Description

`all()`
`any()`
`none()`
	Returns `true` if all, at least one, or, respectively, none of the bits are set.

`count()`	Returns the number of bits that are set.
`operator[]`	Accesses a bit at a given index. No bounds checking is performed.
`test()`	Accesses a bit at a given index. Throws `std::out_of_range` if the given index is out of bounds.
`==`, `!=`	Returns `true` if two `bitset`s are equal or not equal, respectively.
`size()`	Returns the number of bits the `bitset` can hold.

`to_string()`
`to_ulong()`
`to_ullong()`
	Converts a `bitset` to a `string`, `unsigned long`, or, respectively, `unsigned long long`.

Operations

Operation	Description
`flip()`	Flips the values of all the bits
`reset()`	Sets all bits or a bit at a specific position to `false`
`set()`	Sets all bits to `true` or a bit at a specific position to a specific value

In addition, `bitset` supports all bitwise operators: `~`, `&`, `&=`, `^`, `^=`, `|`, `|=`, `<<`, `<<=`, `>>`, and `>>=`.

Container Adaptors

Container adaptors are built on top of other containers to provide a different interface. They prevent you from directly accessing the underlying container and force you to use their special interface. The following three sections give an overview of the available container adaptors—queue, priority_queue, and stack—followed by a section that gives an example and a reference section.

std::queue **<queue>**

A queue represents a container that has first-in first-out (FIFO) semantics. You can compare it to a queue at a night club. A person who arrived before you will be allowed to enter before you.

A queue needs access to the front and the back, so the underlying container must support back(), front(), push_back(), and pop_front(). The standard list and deque support these methods and can be used as underlying containers. The default container is the deque. Here is the template definition of queue:

```
template<class T, class Container = std::deque<T>>
class queue;
```

The complexity for a queue is as follows:

- *Insertion*: O(1) for list as underlying container; amortized O(1) for deque
- *Deletion*: O(1) for list and deque as underlying container
- *Access*: Not possible

std::priority_queue **<queue>**

A priority_queue is similar to a queue but stores the elements according to a priority. The element with highest priority is at the front of the queue. In the case of a night club, VIP members get higher priority and are allowed to enter before non-VIPs.

A priority_queue needs random access on the underlying container and only needs to be able to modify the container at the back, not the front. Therefore, the underlying container must support random access, front(), push_back(), and pop_back(). The vector and deque are available options, with the vector being the default underlying container. Here is the template definition of priority_queue:

```
template<class T,
   class Container = std::vector<T>,
   class Compare = std::less<typename Container::value_type>>
class priority_queue;
```

To determine the priority, elements are compared using a functor object of the type specified as the Compare template type parameter. By default, this is std::less, explained in Chapter 2, which, unless specialized, forwards to operator< of the element

type T. A Compare instance can optionally be provided to the priority_queue constructor; if not, one is default-constructed.

The complexity for a priority_queue is as follows:

- *Insertion:* Amortized O(log(N)) for vector or deque as underlying container
- *Deletion:* O(log(N)) for vector and deque as underlying container
- *Access:* Not possible

std::stack <stack>

A stack represents a container that has last-in first-out (LIFO) semantics. You can compare it to a stack of plates in a self-service restaurant. Plates are added at the top, pushing down other plates. A customer takes a plate from the top, which is the last added plate on the stack.

For implementing LIFO semantics, a stack requires the underlying container to support back(), push_back(), and pop_back(). The vector, deque, and list are available options for the underlying container, with deque being the default one. Here is the template definition of stack:

```
template<class T, class Container = std::deque<T>>
class stack;
```

The complexity for a stack is as follows:

- *Insertion:* O(1) for list as underlying container, amortized O(1) for vector and deque
- *Deletion:* O(1) for list, vector and deque as underlying container
- *Access:* Not possible

Example

The following example demonstrates how to use the container adaptors. The table after the code shows the output of the program when the container, cont, is defined as a queue, priority_queue, or stack, respectively:

```
std::queue<Person> cont;
cont.emplace("Doug", "B", true);
cont.emplace("Phil", "W", false);
cont.emplace("Stu", "P", true);
cont.emplace("Alan", "G", false);
while (!cont.empty())
{
  std::cout << cont.front() << std::endl;  // queue
  // std::cout << cont.top() << std::endl; // priority_queue and stack
  cont.pop();
}
```

queue<Person>	priority_queue<Person>	stack<Person>
Doug B	Stu P[4]	Alan G
Phil W	Doug B	Stu P
Stu P	Phil W	Phil W
Alan G	Alan G	Doug B

Reference

Operation	Description
`emplace()`	*Queue*: Constructs a new element in place at the back *Priority queue*: Constructs a new element in place *Stack*: Constructs a new element in place at the top
`empty()`	Returns `true` if empty, `false` otherwise
`front()` `back()`	*Queue*: Returns a reference to the first or last element *Priority queue*: n/a *Stack*: n/a
`pop()`	*Queue*: Removes the first element from the queue *Priority queue*: Removes the highest-priority element *Stack*: Removes the top element
`push()`	*Queue*: Inserts a new element at the back of the queue *Priority queue*: Inserts a new element *Stack*: Inserts a new element at the top
`size()`	Returns the number of elements
`swap()`	Swaps the contents of two queues or stacks
`top()`	*Queue*: n/a *Priority queue*: Returns a reference to the element with the highest priority *Stack*: Returns a reference to the element at the top

`queue` and `stack` support the same set of non-member functions as the sequential containers: `==`, `!=`, `<`, `<=`, `>`, `>=`, and `std::swap()`. `priority_queue` only supports the `std::swap()` non-member function.

Ordered Associative Containers

There are four ordered associative containers: `std::map`, `multimap`, `set`, and `multiset`, which we explain in turn. We'll not revisit the member functions that are analogous to those of the sequential containers seen earlier (`begin()`, `end()`, `size()`, `clear()`, etc.).

[4] The way `operator<` is defined for `Person` in the Introduction causes the VIP and non-VIP persons in the `priority_queue` to be in reverse alphabetical order: people with an alphabetically higher name get a higher priority.

For this, we refer to the overview of all available functions at the end of the section. We begin with `std::map`, which is explained in some more detail. As always, working with the other associative containers is mostly analogous.

std::map `<map>`

A `map` is a data structure that stores key-value elements, each represented using the `pair` utility class explained in Chapter 2. Each key can be associated with at most one value at any given moment.

When defining a `map`, you need to specify both the key type and the value type. You can immediately initialize a `map` with a braced initializer as well:

```
std::map<Person,int> myMap{ {Person("Jenne"), 1}, {Person("Bart"), 2} };
```

■ **Note** In the preceding statement, the compiler would not be able to deduce the `map`'s two template arguments. The reason is that it cannot deduce the expected element type `std::pair<const Person,int>` from the values in the initializer list.

Iterators for a `std::map<Key, Value>` are bidirectional iterators that point to a `std::pair<const Key, Value>`. For example:

```
auto iter = begin(myMap);  // type of *iter is pair<const Person, int>&
std::cout << "Key=" << iter->first.GetFirstName();    // Key=Bart
std::cout << ", Value=" << iter->second << std::endl; // , Value=2
```

As with all ordered associative containers, a `map`'s elements are stored and enumerated in an order with increasing key values, not in the order in which these elements were inserted. This is why, in the previous example, `begin(myMap)` points to the element with key `Person("Bart")` rather than `Person("Jenne")`.

The advantage of a `map<K,V>` over, say, a `vector<pair<K,V>>` is that a `map` allows you to quickly assign and retrieve the value associated with a given key. One way to do this is using its `[]` operator:

```
myMap[Person("Bart")] = 0;
std::cout << myMap[Person("Bart")] << std::endl;   // 0
```

Note the resemblance to an array: the only apparent difference is that elements are indexed by values of type `Key` rather than by consecutive integers. This is why maps are sometimes also called *associative arrays*.

In the next section, we discuss how to insert new elements. You remove an element with `erase()`, which you can pass either an iterator or simply a key:

```
myMap.erase(Person("Bart"));
```

Inserting in a Map

Table 3-1 lists six ways of inserting a single key-value element into a map. The second, third, and fourth columns indicate some differences between them (these differences are explained later). The final column indicates whether a member with the same name exists for any of the other ordered associative containers.

Table 3-1. *Six Ways of Inserting Elements into a Map. "N/A" Means "Not Applicable", and "?" Denotes "Unspecified"*

Basic Usage	Overwrites	Steals	Hint	Maps Only
`map[key] = value;`	Yes	N/A	No	Yes
`map.insert({key,value});` `map.insert(std::pair(key,value));`	No	?	Yes	No
`map.insert_or_assign(key,value);` C++17	Yes	N/A	Yes	Yes
`map.emplace(key,value);`	No	?	No	No
`map.emplace_hint(hint,key,value);`	No	?	Only	No
`map.try_emplace(key,value);` C++17 `map.try_emplace(key,args...);` C++17	No	No	Yes	Yes

The Square Brackets Operator

You can use `operator[]` not only to access existing elements, as shown earlier, but also to add new elements. If there is no value associated yet with the given key, `operator[]` inserts a new element that associates this key with a new, default-constructed value and then returns a reference to that value. This means you could add a new element to `myMap` from our earlier example as follows:

```
myMap[Person("Peter")] = 3;
```

■ **Tip** The fact that `operator[]` may modify the container implies you cannot use it on a `const map`. To access elements from a `const map`, you can use the `at()` member instead. Take care though: when the given key is not associated with a value, `at()` throws a `std::out_of_range` exception. Another option is the `find()` method, which returns either an iterator to the requested key-value `pair` or the `end()` iterator.

insert() and insert_or_assign()

Your second option to add elements to a `map` is with its `insert()` method, like this (remember: an *element* of a `map` is a key-value *`pair`*):

```
myMap.insert(std::pair(Person("Marc"), 4));
```

Or, using a braced initializer for short:

```
myMap.insert({ Person("Marc"), 4 });
```

Other overloads of `insert()` allow you to add multiple elements at once. You provide these elements either as an iterator range or as an initializer list. `insert()` is the only member of Table 3-1 that allows you to insert multiple elements at once.

`insert()` never overwrites a value previously associated with a given key. Suppose `Person("Marc")` is still mapped to 4 from before. Then:

```
myMap.insert({ Person("Marc"), 5 });              //5 is discarded
std::cout << myMap[Person("Marc")] << std::endl;  // still 4!
```

To update existing mappings, you can use either the square brackets operator or `insert_or_assign()`:

```
myMap.insert_or_assign(Person("Marc"), 5);
std::cout << myMap[Person("Marc")] << std::endl;  // 5
```

Notice how you do not pass a `std::pair` element to `insert_or_assign()`. Instead you pass a key followed by a value. The following example illustrates this:

```
std::map<int, Person> inverseMap;
inverseMap.insert({ 3, Person("Peter", "Van Weert") });
inverseMap.insert_or_assign(5, Person("Marc", "Gregoire"));
```

emplace() and try_emplace()

There is also an `emplace()` method that allows you to construct a new key-value pair in place. You pass it the same arguments you would when constructing a key-value `pair`. In practice, this is again mostly a key and a value. For example:

```
inverseMap.emplace(6, Person("Christophe", "Pichaud"));
```

To avoid the creation of any temporary objects with `emplace()`, you could use *piecewise construction* and `forward_as_tuple()`, as explained in Chapter 2 (remember: all arguments of `emplace()` are forwarded to a constructor of `std::pair`):

```
inverseMap.emplace(std::piecewise_construct, std::forward_as_tuple(6),
                   std::forward_as_tuple("Christophe", "Pichaud"));
```

If you instead use `try_emplace()`, in-place construction of the value object is done automatically. The parameters to this method are again slightly different: you pass it a key followed by the arguments required to construct a value:

```
inverseMap.try_emplace(6, "Christophe", "Pichaud");
```

The second difference between `emplace()` and `try_emplace()` is even more subtle. For keys that *are* already mapped, `try_emplace()` is the only insertion function guaranteed to do absolutely *nothing*. The following example will clarify. Suppose that the key 6 is already mapped from before. Ask yourself: what then happens to the `powers` object in the following snippet?

```
Person powers("Mark", "Powers");
inverseMap.emplace(6, std::move(powers));
```

`emplace()`, like `insert()`, never overwrites, so ultimately the addition is discarded. That is clear. But the standard does not specify whether the contents of `powers` are moved first (say, into some temporary `pair` element), before the insertion is eventually discarded. It is said that `emplace()` and `insert()` may "steal" a value if the given key is already mapped. If you use `try_emplace()`, the standard guarantees that `powers` will *not* be stolen; that is, its contents will *not* be moved.

Hints

Most insertion members (see Table 3-1) allow for an optional 'hint' iterator to be passed as the first argument. If this 'hint' points to the position right *after* the position where the new element should be inserted, the insertion is done in amortized constant time. Otherwise, insertion is logarithmic.

Return Values

If no hint is used, members that insert a single element in a `map` generally return a `std::pair` consisting of these two values (in the given order):

- An iterator pointing either to the newly inserted element (a new key-value pair) or to the already-existing element that prevented the insertion
- A Boolean that is `true` if a new element was inserted, or `false` otherwise

Here is an example, where we use a structured binding (a syntax that is new in C++17) to decompose the `std::pair`:

```
auto [iter, inserted] = inverseMap.insert({ 6, Person("Marc") });
if (!inserted)
   std::cout << "Not inserted. Existing value: " << iter->second;
```

If a hint is used, only the iterator is returned. And if multiple values are inserted at once, the return type is always void.

std::multimap <map>

A multimap is mostly analogous to a map, except that it allows the same key to be associated with multiple values at once. Elements are sorted on their key, of course, and elements with the same key are and remain stored in the order of their insertion:

```
std::multimap<std::string,int> myMulti;
myMulti.insert({ "someKey", 2 });
myMulti.insert({ "sameKey", 5 });
myMulti.insert({ "sameKey", 2 });
std::cout << myMulti.size() << ' ' << myMulti.begin()->second;  // 3 5
```

A multimap has no square brackets operator and no insert_or_assign() or try_emplace() members. The insertion members do not return a pair, but simply an iterator to the inserted element.

std::set and std::multiset <set>

A set is similar to a map, but it does not store pairs, only unique keys without values (this is how the standard defines it, and we will as well; some may prefer to think of it as values without keys though). A multiset supports duplicate keys.

There is only one template type parameter: the key type. This template argument can be deduced if initial values are provided. The insert() method takes a single key instead of a pair. For example:

```
std::set mySet{ 3,2,1};   // std::set<int> deduced
mySet.insert(2);
mySet.insert(6);
std::cout << mySet.size() << ' ' << *mySet.begin();    // 4 1
```

There are overloads of insert() similar to those for map and multimap, and similar emplacement functions as well. Removal is again done through erase().

An iterator for a set or multiset is bidirectional and points to the actual key, not to a pair. Keys are always sorted.

Order of Elements

The ordered associative containers store their elements in an ordered fashion. By default, std::less<Key> is used for this ordering, which, unless specialized, relies on operator< of the Key type. You can change the comparison functor type by specifying a Compare template type parameter. Here are near-complete template definitions of all four ordered associative containers:

```
template<typename Key, typename Value, class Compare = std::less<Key>>
class map;
```

```
template<typename Key, typename Value, class Compare = std::less<Key>>
class multimap;

template<typename Key, typename Compare = std::less<Key>>
class set;
template<typename Key, typename Compare = std::less<Key>>
class multiset;
```

If no `Compare` functor instance is passed to the constructors of these containers (as in all examples thus far), one is default-constructed.

■ **Tip** The preferred functors for use with ordered associative containers are the so-called *transparent operator functors* (see Chapter 2)—for example, `std::less<>` (short for `std::less<void>`)—because this improves performance for so-called *heterogeneous lookups*. A `map` from `string` to `int` with a transparent operator functor, for example, is declared as follows:

```
std::map<std::string,int,std::less<>> myStringMap;
```

A `map` of this type will be more efficient than a regular `std::map<std::string,int>` when, for instance, string literals (or `string_views`: see Chapter 6) are used as keys in lookups: `std::less<>` then avoids the creation of temporary `std::string` objects.

Searching

If you want to find out whether a certain key is in an associative container, you can use these:

- `find()`: Returns an iterator to the found element (a key-value pair for maps) or the end iterator if the given key is not found.
- `count()`: Returns the number of keys matching the given key. For `map` or `set`, this can only be 0 or 1, whereas for `multimap` or `multiset`, this can be larger than 1.
- `lower_bound()`: Returns an iterator that points to the first element whose key is *not less* than a given key. This can be the end iterator if the key is not present in the container.
- `upper_bound()`: Returns an iterator that points to the first element whose key is *greater* than a given key. Same as `lower_bound()` if there is no element with the given key.
- `equal_range()`: Returns a `std::pair` of two iterators: `lower_bound()` and `upper_bound()`. This corresponds to the half-open range of elements with the given key if these exist.

■ **Tip** The result of `upper_bound()`—and thus that of `lower_bound()` as well for unmapped keys—is a suitable hint iterator for the insertion members we discussed earlier.

Moving Nodes Between Containers C++17

Ordered associative containers are generally implemented using a node-based data structure (typically red-black trees). The following code efficiently moves one such node from `myMap` to `myOtherMap`. No elements are copied or even moved in the process; only links between nodes are updated:

```
auto node = myMap.extract(Person("Marc")); // type std::map::node_type
if (node)                                   // or: !node.empty()
  myOtherMap.insert(std::move(node));    // would ignore empty node
```

The node handles returned by `extract()` cannot be copied, only moved.

A node from a `map` can be inserted into a `multimap` as well, and vice versa. The same holds for transferring nodes between `set`s and `multiset`s. The only requirement is that the target container has the exact same element type as the source container. The type of the comparison functors need not be the same.

The node-based overloads of `insert()` for `std::map` and `set` do not return a `pair`, but a `struct` of type *`Container`*`::insert_return_type`. This `struct` has the following three public data members (in that order):

- `position`: An iterator pointing to either the inserted element, the element that prevented the insertion, or `end()` if the given node handle was empty
- `inserted`: `true` if a new element was inserted; `false` otherwise
- `node`: The value of the (moved) node handle if no insertion took place

Merging Containers C++17

You can merge two associative containers with their `merge()` member functions. You can merge any two ordered associative containers, as long as both have the exact same element type. This implies that you can never merge a `set` into a `map`, but that you can, for instance, merge a `multiset` into a `set`:

```
std::multiset src{ 1, 2, 4 };
std::set dst{ 2, 3 };
dst.merge(src);
// src == { 2 }
// dst == { 1, 2, 3, 4 }
```

The preceding example also illustrates that elements that cannot be moved into the destination container (because that container does not allow for duplicate keys) remain behind in the source container.

Complexity

The complexity for all four ordered associative containers is the same:

- *Insertion*: Amortized constant if you provide a correct hint (one past the insertion point); O(log(N)) otherwise.
- *Deletion*: O(log(N))
- *Access*: O(log(N))

Reference

The following subsections give an overview of all the operations supported by `map` (M), `multimap` (MM), `set` (S), and `multiset` (MS), divided into categories.

Iterators

All ordered associative containers support the same set of iterator-related methods as supported by the `vector` container: `begin()`, `end()`, `cbegin()`, `cend()`, `rbegin()`, `rend()`, `crbegin()`, and `crend()`.

Size

All associative containers support the following methods:

Operation	Description
`empty()`	Returns `true` if the container is empty, `false` otherwise
`max_size()`	Returns the maximum number of elements that can be stored
`size()`	Returns the number of elements

Access and Lookup

Operation	M	MM	S	MS	Description
`at()`	■	□	□	□	Returns a reference to an element with the given key. If the given key does not exist, a `std::out_of_range` exception is thrown.
`operator[]`	■	□	□	□	Returns a reference to an element with the given key. It default-constructs an element with the given key if one does not exist already.
`count()`	■	■	■	■	Returns the number of elements that match a given key.
`find()`	■	■	■	■	Finds an element matching a given key.
`lower_bound()`	■	■	■	■	Returns an iterator to the first element with a key not less than a given key.
`upper_bound()`	■	■	■	■	Returns an iterator to the first element with a key greater than a given key.
`equal_range()`	■	■	■	■	Returns a range of elements that match a given key as a pair of iterators. The range is equivalent to calling `lower_bound()` and `upper_bound()`.

Modifiers

All associative containers support the following methods:

Operation	Description
`clear()`	Clears the container.
`emplace()`	Constructs a new element in place. Does not overwrite existing elements.
`emplace_hint()`	Same as `emplace()`, but with a hint an implementation may use to speed up insertion. Other insertion members are overloaded to allow for an optional hint as well.
`erase()`	Removes an element at a specific position, a range of elements, or all elements matching a given key.
`extract()`	Extracts a node for efficient insertion into another container.
`insert()`	Inserts new elements or nodes extracted from another container. Never overwrites existing elements.
`merge()`	Merges a given container into the container. If duplicate keys are not allowed in the destination container, elements with duplicate keys remain behind in the source container.
`swap()`	Swaps the contents of two containers.

`std::map` additionally supports the following insertion alternatives. None of the other ordered associative containers support these (C++17).

Operation	Description
`insert_or_assign()`	Inserts a new element, or overwrites the value of an existing key-value association.
`try_emplace()`	Constructs a new key-value pair in place. Does nothing if the given key is already mapped.

Observers

All associative containers support the following observers:

Operation	Description
`key_comp()`	Returns the functor used to compare keys
`value_comp()`	Returns the functor used to compare key-value pairs based on their keys

Non-member Functions

All ordered associative containers support a similar set of non-member functions as the sequential containers: `operator==`, `!=`, `<`, `<=`, `>`, `>=`, `std::swap()`, `std::size()`, and `std::empty()`.

Unordered Associative Containers

`<unordered_map>`, `<unordered_set>`

The Standard Library defines the following four unordered associative containers: `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`. They are completely analogous to their ordered counterparts (`map`, `multimap`, `set`, and `multiset`), except that they do not order their elements but instead store them in buckets in a so-called *hash map*.

Hash Map

A *hash map* or *hash table* is an efficient data structure storing its elements in buckets.[5] Conceptually, the map contains an array of pointers to buckets, which are in turn arrays or linked lists of elements. Through a mathematical formula called *hashing*, a hash integer number is calculated, which is then transformed into a bucket index. Two elements resulting in the same bucket index are stored inside the same bucket.

A hash map allows for very fast retrieval of elements. To retrieve an element, calculate its hash value, which results in the bucket number. If there are multiple elements in that bucket, a quick (generally linear) search is performed in that single bucket to find the right element.

Template Type Parameters

The unordered associative containers allow you to specify your own hasher and your own definition of how to decide whether two keys are equal by specifying extra template type parameters. Here are the near-complete template definitions for all unordered associative containers:

```
template<typename Key, typename Value, class Hash = std::hash<Key>,
class KeyEqual = std::equal_to<Key>> class unordered_map;
template<typename Key, typename Value, class Hash = std::hash<Key>,
class KeyEqual = std::equal_to<Key>> class unordered_multimap;

template<typename Key, class Hash = std::hash<Key>,
class KeyEqual = std::equal_to<Key>> class unordered_set;
template<typename Key, class Hash = std::hash<Key>,
class KeyEqual = std::equal_to<Key>> class unordered_multiset;
```

■ **Caution** The hash function should be compatible with the equality function in the following sense: two keys that are equal should map to the same hash value.

Hash Functions

If too many keys result in the same hash (bucket index), the performance of a hash map deteriorates. In the worst case, all elements end up in the same bucket and all lookup and insertion operations become linear.

---

[5] Technically, you could easily implement a hash map without buckets: e.g., using so-called *open addressing*. The way the standard unordered containers are defined, though, strongly suggests the use of a *separate chaining method*, which is therefore what we describe here.

The Standard Library provides the following `std::hash` template (the base template is defined in `<functional>` but is also included in the `<unordered_xxx>` headers):

```
template<typename T> struct hash;
```

Specializations are provided for several types, such as `bool`, `char`, `int`, `long`, `double`, and `std::string`. If you want to calculate a hash of your own object types, you can implement your own hashing functor class. However, we recommend that you implement a specialization of `std::hash` instead.

The following is an example of how you could implement a `std::hash` specialization for the `Person` class defined in the Introduction. It uses the standard `std::hash` specialization for `string` objects to calculate the hash of the first and last name. Both hashes are then combined by a XOR operation:

```
namespace std {
   template<> struct hash<Person> {
      // The requirement to add the following two type aliases
      // is deprecated in C++17 (and expected to be removed in C++20)
      using argument_type = Person;
      using result_type = std::size_t;

      result_type operator()(const argument_type& p) const {
         auto firstNameHash(std::hash<std::string>()(p.GetFirstName()));
         auto lastNameHash(std::hash<std::string>()(p.GetLastName()));
         return firstNameHash ^ lastNameHash;
      }
   };
}
```

Simply XORing values generally does not give sufficiently randomly distributed integers, but if both operands are already hashes, it can be considered acceptable. Details of writing proper hash functions fall outside the scope of this book.

■ **Tip** Although adding types or functions to the `std` namespace is disallowed, adding template specializations is perfectly legal. Note also that the recommendation we made in Chapter 2 to specialize `std::swap()` in the type's own namespace does not extend to `std::hash`. Because `std::hash` is a class rather than a function (like `swap()`), ADL does not apply (see the discussion in Chapter 2).

Complexity

The complexity for all four unordered associative containers is the same:

- *Insertion:* O(1) on average, O(N) worst case
- *Deletion:* O(1) on average, O(N) worst case
- *Access:* O(1) on average, O(N) worst case

A good hash function is key in achieving O(1) performance. The higher the probability is that a hash function maps different keys onto the same hash value, the further the performance of an unordered associative container degrades to O(N).

■ **Tip** Benchmarks have shown that the unordered associative containers are mostly faster than their ordered counterparts, which makes them good defaults when you are doubting between an ordered and an unordered associative container. Unless, of course, ordered traversal is a requirement, or defining a good hash function is too difficult (defining a less than operator is often easier). This rule of thumb notwithstanding: the only way to know for sure which container performs best for your use case is to benchmark the different options with realistic data and operations.

Reference

All unordered associative containers support the same methods as the ordered associative containers, except reverse iterators, `lower_bound()`, and `upper_bound()`. The following subsections give an overview of all additional operations supported by `std::unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`, divided into categories.

Observers

All unordered associative containers support the following observers:

Operation	Description
`hash_function()`	Returns the hash function used for hashing keys
`key_eq()`	Returns the function used to perform an equality test on keys

Bucket Interface

All unordered associative containers support the following bucket interface:

Operation	Description

`begin(int)`
`end(int)`
`cbegin(int)`
`cend(int)`
	Returns an iterator to the first or one past the last element in the bucket with given index

`bucket(key)`	Returns the index of the bucket for a given key
`bucket_count()`	Returns the number of buckets
`bucket_size(int)`	Returns the number of elements in the bucket with a given index
`max_bucket_count()`	Returns the maximum number of buckets that can be created

Hash Policy

All unordered associative containers support the following hash policy methods:

Operation	Description
`load_factor()`	Returns the average number of elements in a bucket.
`max_load_factor()`	Returns or sets the maximum load factor. If the load factor exceeds this maximum, more buckets are created.
`rehash()`	Sets the number of buckets to a specific value and rehashes all current elements.
`reserve()`	Reserves a number of buckets to accommodate a given number of elements without exceeding the maximum load factor.

Non-member Functions

Unordered associative containers support `operator==`, `operator!=`, `std::swap()`, `std::size()`, and `std::empty()` as non-member functions.

Allocators <memory>

An *allocator* is a functor responsible for the allocation and deallocation of memory. Here is how you could, in principle, use one directly:

```
std::allocator<Person> allocator;   // Default allocator
Person* p = allocator.allocate(1);  // Person constructor not invoked
new(p) Person("Basil", "Fawlty");   // Initialize using placement new
p->~Person();                       // Memory not deallocated yet
allocator.deallocate(p, 1);
```

To allocate enough uninitialized memory to hold an entire array of elements, you simply pass the desired array size to `allocate()` / `deallocate()` (instead of `1` in the example). You can find a series of algorithms to subsequently populate and manipulate such blocks of uninitialized memory in Chapter 4.

You generally do not use an allocator directly, however. In the Standard Library, various types use them to manage their dynamic memory. By default, they all use `std::allocator<T>`, but you can override this.

All containers except `array` and `bitset`, for instance, support an optional template type parameter we have not shown yet—one that allows you to specify the type of allocator it should use. For example, the complete definition of the `vector` template is as follows:

```
template<class T, class Allocator = allocator<T>>
class vector;
```

An analogous template parameter exists for `std::basic_string` (Chapter 6) as well. Other Standard Library types allow you to pass an allocator object as an optional constructor argument. Examples include `function` and `shared_ptr` from Chapter 2, and `promise` and `packaged_task` from Chapter 7. Note that to construct a `shared_ptr` with a nondefault allocator, you should use `std::allocate_shared()`.

Mostly, though, the default allocator suffices. Only on rare occasions you want to control how memory is allocated. If the need does arise, we recommend you use an existing allocator type provided either by the standard (see next subsection) or by a third-party library (such as Boost). Writing your own custom allocator is possible, but this is an advanced topic. We therefore do not cover this here.

Polymorphic Allocators C++17 <memory_resource>

The allocator that a container uses is part of its type. This sometimes impedes the use of custom allocators: a `vector<T, allocator1>`, for instance, cannot be used for functions that expect either a `vector<T>` or a `vector<T, allocator2>`. To address this design defect, C++17 has introduced `std::pmr::polymorphic_allocator<T>`.

You can construct a `polymorphic_allocator` with a pointer to an object of any type derived from `std::pmr::memory_resource`, after which the allocator invokes virtual

functions on this memory_resource. The allocation mechanism is thus determined using runtime polymorphism rather than template instantiation.

A default-constructed polymorphic_allocator uses the default memory resource it obtains from std::pmr::get_default_resource(). This global default can be changed (in a thread-safe manner) using std::pmr::set_default_resource(). By default, this default is set to the result of std::pmr::new_delete_resource(), which is a pointer to a global memory_resource instance that simply uses new and delete.

std::pmr::null_memory_resource() returns a pointer to a second, equally trivial global memory_resource: one that throws bad_alloc() for every allocation. We give an example of when this may be useful in the upcoming section on monotonic buffers.

Type Aliases

Because std::vector<T, std::pmr::polymorphic_allocator<T>> is quite a mouthful, the <vector> header defines an alias for this: std::pmr::vector<T>. Analogous aliases are added to the std::pmr namespace for all container types, all basic_string aliases (Chapter 6), and all match_result aliases (Chapter 6), all by their respective headers.

Monotonic Buffers

A std::pmr::monotonic_buffer_resource allocates sequentially from its latest buffer. If a buffer is depleted, a new (generally geometrically larger) buffer is allocated from an *upstream memory resource* (by default std::pmr::get_default_resource()). The type gets its name from the fact that it never really deallocates individual blocks. Its deallocation function itself does nothing at all. That way, both allocation and deallocation can be kept extremely fast.

An initial buffer may be provided to the constructor. Here is an overview of the available constructors:

```
monotonic_buffer_resource([memory_resource* upstream])
monotonic_buffer_resource(std::size_t size[, memory_resource* upstream])
monotonic_buffer_resource(void* buffer, std::size_t size
                                        [, memory_resource* upstream])
```

For the middle constructors, size represents a minimum size for the initial, yet-to-be-allocated first buffer.

Other members are release(), which deallocates all owned buffers, and upstream_resource(), which returns a pointer to the upstream memory resource.

■ **Caution** Besides any initial buffer passed to its constructors, all buffers are owned by the monotonic_buffer_resource. It will deallocate these upon destruction (or when release() is called). This memory resource must therefore outlive all objects that are using memory allocated from these buffers.

The idea is to use this `memory_resource` to quickly allocate a collection of objects and then discard them again, all at once. Another key use is to turn dynamic allocation into static allocation. An example will clarify. Suppose we are working in a context where we know that a string will never be longer than a few hundred characters. Then you can allocate its memory on the stack while still using the familiar `std::string` class, as follows:

```
char buffer[300];   // Static allocation on the stack
std::pmr::monotonic_buffer_resource resource(
   buffer, std::size(buffer), std::pmr::null_memory_resource()
);
std::pmr::string short_string(&resource);
// ... fill and manipulate short_string
```

This avoids the cost of dynamic allocation, which would make a difference if this code either executes many times or is used in an environment where resources are particularly scarce.

Note also that `null_memory_resource()` is set as the upstream allocator. `bad_alloc()` will therefore be raised if the stack-allocated buffer does not suffice. Doing so is optional and would be done for debugging purposes: with the default upstream allocator, the `monotonic_buffer_resource` would otherwise silently allocate a new buffer, which might nullify the intended performance optimization.

■ **Caution** It is often best to add some safety margin to your buffer sizes: a default-constructed `std::string`, for instance, may already allocate some memory (to store the empty string). Furthermore, `reserve()` and similar members are allowed to allocate more than requested.

Memory Pools

The `memory_resources` discussed in this section manage a collection of *memory pools*, each serving requests for a different (maximum) size. Each of these pools consists of a collection of *chunks*, in turn divided in equally sized *blocks*. Once a pool's latest chunk is depleted, a new (generally geometrically larger) chunk is acquired from an *upstream memory resource*. Allocation requests whose size exceed the largest block size are forwarded to this same upstream allocator.

The two `memory_resources` in question are `std::pmr::synchronized_pool_resource` and `unsynchronized_pool_resource`. The former is thread-safe and optimized for concurrent use, whereas the latter is more efficient if used from a single thread.

Both types offer constructors of the following forms:

```
..._pool_resource([memory_resource* upstream])
..._pool_resource(const pool_options& options[, memory_resource* upstream])
```

The default upstream `memory_resource` is again determined by `std::pmr::get_default_resource()`. The other optional parameter is of type `std::pmr::pool_options`, a struct consisting of these two `size_t` variables:

- `max_blocks_per_chunk`: Maximum number of blocks per chunk. Implementations may stop growing chunk sizes at less blocks as well or use different maximum chunk sizes for different pools.
- `largest_required_pool_block`: The allocation size that the `memory_resource` should at least be able to service without allocating from its upstream allocator. The actual block size used may be larger.

If either value is zero or if the `pool_options` argument is omitted altogether, implementation-specific default values are used.

Other members are `release()`, which deallocates all buffers (same as the destructor), and the obvious getters `upstream_resource()` and `options()`.

Allocators for Multilevel Containers **<scoped_allocator>**

The `std::scoped_allocator_adaptor<OuterAlloc, InnerAllocs...>` template can be used as the allocator type of a container of containers. It consists of an *outer allocator* of type `OuterAlloc` and zero or more *inner allocators* of potentially different types. References to all adapted allocator objects can be provided upon construction. If a container is declared with `scoped_allocator_adaptor` as its allocator type, it uses its outer allocator to allocate its elements. If these elements are containers themselves, they in turn use the first of the `InnerAllocs`, and so on. If there are more levels of containers than allocator types, the last of the inner allocators is used for the remaining innermost containers.

CHAPTER 4

Algorithms

The previous chapter discusses the containers provided by the Standard Library to store data. Orthogonally to these, the library offers numerous algorithms to process this or other data. Algorithms are independent of containers: they do their work solely based on iterators and can therefore be executed on any range of elements as long as suitable iterators are provided.

This chapter starts with a brief definition of input/output iterators, followed by a detailed overview of all available algorithms organized by functionality. The chapter ends with a discussion of iterator adaptors.

Input and Output Iterators

The previous chapter briefly explains the different categories of iterators offered by containers: forward, bidirectional, and random access. Two more iterator categories are used in the context of algorithms, which have fewer requirements compared to the other three. Essentially:

- *Input iterator*: Must be dereferenceable to read elements. Other than that, only the ++, ==, and != operators are required.
- *Output iterator*: Only ++ operators are required, but you must be able to write elements to them after dereferencing.

For both, it also suffices that they provide single-pass access. That is, once incremented, they may in principle invalidate all previous copies of them. Two corresponding iterator tags, as discussed in Chapter 3, are provided for these categories as well: `std::input_iterator_tag` and `output_iterator_tag`.

All iterators returned by the standard containers, as well as pointers into C-style arrays, are valid input iterators. They are valid output iterators as well, as long as they do not point to `const` elements.

P. Van Weert and M. Gregoire, *C++17 Standard Library Quick Reference*,
https://doi.org/10.1007/978-1-4842-4923-9_4

General Guidelines

First and foremost: use standard algorithms instead of self-written loops whenever possible, because they are often more efficient and are far less error-prone. Also, and especially after the introduction of lambda expressions, the use of algorithms mostly results in far shorter, readable, self-explanatory code.

Whenever a container offers a specialized member function equivalent to an algorithm (see Chapter 3), though, these are even more efficient and should therefore be preferred over the generic algorithms. In the algorithm descriptions that follow, we always list these alternatives.

Our remaining guidelines concern the two types of arguments you generally pass to an algorithm: iterators, and more specifically the *elements* these iterators point to, and function-like *callables*.

Algorithm Arguments

Many algorithms either move or swap elements. Without proper move and/or swap functions, these algorithms fall back to copying. For optimal performance you should therefore always consider implementing specialized move and/or swap functions for nontrivial custom data types (unless of course the implicitly generated members suffice: remember the Rule of Zero!). Types offered by the Standard Library always provide move and swap functions where appropriate. We refer to Chapter 2 for more information regarding move semantics and swap functions.

Most algorithms also accept one or more *callables*—typically unary or binary operations or predicates, as defined in the next section. Such callables can be either lambda expressions, function pointers, or function objects (function objects are discussed in Chapter 2). For most algorithms the following restrictions apply:

- Callables are not allowed to modify any elements through the references they receive as input arguments.
- Algorithms may generally copy the callables they receive as often as they want. Passing a so-called *stateful functor* may therefore not always work as expected: whenever a function operator updates a member variable, it may be updating the state of some *copy* of the original functor. One solution is to wrap such functors inside a `std::reference_wrapper` (see Chapter 2) before passing them to an algorithm.

The only algorithms for which these restrictions do not apply are `for_each()` and `for_each_n()`: as noted also later, these algorithms allow elements to be updated by their callables and are guaranteed to never copy their callables (except when parallel execution is used, as introduced near the end of this chapter).

Terminology

The following terms and abbreviations are used for types in the definitions of algorithms:

- *Function*: Callable—that is, lambda expression, function object, or function pointer.
- *InIt, OutIt, FwIt, BidIt, RanIt*: Input, output, forward, bidirectional, or random access iterator.
- *UnaOp, BinOp*: Unary or binary operation—that is, a callable accepting one resp. two arguments.
- *UnaPred, BinPred*: Unary or binary predicate, with a predicate being an operation that returns a Boolean.
- *Size*: A type representing a size—for example, a number of elements.
- *DiffType*: A type representing a distance between two iterators.
- *T*: An element type.
- *Compare*: A function object to be used to compare elements. If not specified, `operator<` is used. The function object accepts two parameters and returns `true` if the first argument is less than the second, `false` otherwise. The ordering imposed must be a strict weak ordering, just as with the default `operator<`.

Algorithms **<algorithm>**

This section gives an overview of all available algorithms in the `<algorithm>` header, organized into subsections according to functionality. Algorithms offered by other headers are listed in later sections.

Applying a Function to a Range

`Function for_each(InIt first, InIt last, Function function)`

Invokes the given unary function for each element in the range `[first, last)` and returns `std::move(function)`. `function` is invoked with dereferenced iterators from the input range. It is typically a `void` function (though not necessarily: returned values are simply ignored) and is allowed to modify the elements it receives as input arguments. `for_each()` does not copy `function`.

Note that when iterating sequentially over an entire container or C-style array, a range-based `for` loop is more convenient. `std::for_each()` is mostly useful for applying a function either to subranges or to all elements in parallel (see later in this chapter).

```
InIt for_each_n(InIt first, Size n, Function function) C++17
```

Similar to `for_each()`, except that it operates on the range `[first, first + n)`, and returns the end iterator, `first + n`.

```
OutIt transform(InIt first1, InIt last1, OutIt target, UnaOp operation)
OutIt transform(InIt1 first1, InIt1 last1, InIt2 first2,
                OutIt target, BinOp operation)
```

Transforms all elements in a range `[first1, last1)` and stores the results in a range starting at `target`. The output iterator `target` is allowed to be equal to `first1` or `first2` to perform an in-place transformation. For the first version, a unary operation is invoked for each element. For the second version, a binary operation is similarly performed on corresponding elements from both input ranges. Let *length* = `(last1 - first1)`, then the binary operation is executed on pairs `(*(first1 + n), *(first2 + n))` with $0 \le n <$ *length*. Both versions write the return values of these invocations to the target range, in order. Unlike `for_each()` and `for_each_n()`, the given `operation` is not allowed to modify the elements it receives as input arguments. Both versions return the end iterator of the target range, so `(target + length)`.

Example

The following example uses `transform()` first to double all the elements in a `vector` using a lambda expression, and then again to negate the elements using a standard function object. Next, it increments all numbers with `for_each()`. And finally, it outputs all elements to the console using `for_each_n()`. This code snippet additionally needs `<functional>` for `std::negate`:

```
std::vector vec{ 1,2,3,4,5,6 };

std::transform(cbegin(vec), cend(vec), begin(vec),
   [](auto& element) { return element * 2; });

std::transform(cbegin(vec), cend(vec), begin(vec), std::negate<>());

std::for_each(begin(vec), end(vec), [](int& i) { i += 1; });

std::for_each_n(cbegin(vec), size(vec),
   [](const auto& element) { std::cout << element << " "; });
```

The output is as follows:

```
-1 -3 -5 -7 -9 -11
```

Checking for the Presence of Elements

```
bool all_of(InIt first, InIt last, UnaPred predicate)
bool none_of(InIt first, InIt last, UnaPred predicate)
bool any_of(InIt first, InIt last, UnaPred predicate)
```

Returns `true` if all, none, or respectively at least one of the elements in the range `[first, last)` satisfies a unary `predicate`. If the range is empty, `all_of()` and `none_of()` return `true`, and `any_of()` returns `false`.

```
DiffType count(InIt first, InIt last, const T& value)
DiffType count_if(InIt first, InIt last, UnaPred predicate)
```

Returns the number of elements in `[first, last)` that are equal to a given `value`, or that satisfy a unary `predicate`.

[Alternatives: all associative containers have a `count()` member.]

Example

The following example demonstrates the use of `all_of()` to check whether all elements are even:

```
std::vector vec{ 1,2,3,4,5,6 };
bool allEven = std::all_of(cbegin(vec), cend(vec),
 [](auto& element) { return element % 2 == 0; });  // false
```

Finding Elements

The algorithms in this section search for the first element in a range that satisfies some requirement. To obtain not just a single element but all elements that satisfy such a requirement, you can, for instance, use `std::copy_if()` instead. This algorithm is explained later in this chapter. Use reverse iterators to scan for the last element rather than the first.

```
InIt find(InIt first, InIt last, const T& value)
InIt find_if(InIt first, InIt last, UnaPred predicate)
InIt find_if_not(InIt first, InIt last, UnaPred predicate)
```

Searches all elements in the range `[first, last)` for the first element that is equal to a `value`, satisfies a unary `predicate`, or does not satisfy a `predicate`. Returns an iterator to the element found, or `last` if none is found.

[Alternatives: all associative containers have a `find()` member.]

```
InIt find_first_of(InIt first1, InIt last1,
                   FwIt first2, FwIt last2[, BinPred predicate])
```

Returns an iterator to the first element in `[first1, last1)` that is equal to an element in `[first2, last2)`. Returns `last1` if no such element is found or if `[first2, last2)` is empty. If a binary predicate is given, it is used to decide about equality of elements between the two ranges.

```
FwIt adjacent_find(FwIt first, FwIt last[, BinPred predicate])
```

Returns an iterator to the first element of the first pair of adjacent elements in the range `[first, last)` that are equal to each other or match a binary `predicate`. Returns `last` if no suited adjacent elements are found.

Example

The following code snippet uses the `find_if()` algorithm to find a person called Waldo in a list of people:

```
auto people = { Person("Wally"), Person("Wilma"), Person("Wenda"),
                Person("Odlaw"), Person("Waldo"), Person("Woof") };
auto iter = std::find_if(begin(people), end(people),
   [](const Person& p) { return p.GetFirstName() == "Waldo"; });
```

Finding Min/Max Elements

In Chapter 1, we introduced the `min()`, `max()`, and `minmax()` functions defined by the `<algorithm>` header. Unlike most other `<algorithm>` functions, these do not operate on iterator ranges. Instead they operate either on two elements or on an initializer list. To find the smallest and/or largest elements of a range, you can use the following algorithms instead:

```
FwIt min_element(FwIt first, FwIt last[, Compare comp])
FwIt max_element(FwIt first, FwIt last[, Compare comp])
pair<FwIt, FwIt> minmax_element(FwIt first, FwIt last[, Compare comp])
```

Returns an iterator to the minimum, an iterator to the maximum, or a `pair` containing an iterator to both the minimum and maximum element in a range `[first, last)`. Returns `last` or `pair(first, first)` if the range is empty.

Binary Search

All of the following algorithms require that the given range [`first`, `last`) is sorted.[1] If this precondition is not met, the algorithms' behavior is undefined:

```
bool binary_search(FwIt first, FwIt last, const T& value[, Compare comp])
```

Returns `true` if there is an element equal to `value` in the range [`first`, `last`).

```
FwIt lower_bound(FwIt first, FwIt last, const T& value[, Compare comp])
FwIt upper_bound(FwIt first, FwIt last, const T& value[, Compare comp])
```

Returns an iterator to the first element in [`first`, `last`) that does not compare less than `value` for `lower_bound()` and to the first that compares greater than `value` for `upper_bound()`. When inserting in a sorted range, both are suitable positions to insert `value`, provided insertion happens before the iterator (as with the `insert()` method of sequential containers: see the next "Example" subsection).

[Alternatives: all ordered associative containers have `lower_bound()` and `upper_bound()` members.]

```
pair<FwIt, FwIt> equal_range(FwIt first, FwIt last,
                             const T& value[, Compare comp])
```

Returns a `pair` containing the lower and upper bounds.

[Alternatives: all associative containers, including the unordered ones, have an `equal_range()` member.]

Example

The following code snippet demonstrates how to insert a new value into a `vector` at the correct place to keep the elements sorted:

```
std::vector vec{ 11,22,33 };
const int valueToAdd = 18;
auto lower = std::lower_bound(cbegin(vec), cend(vec), valueToAdd);
vec.insert(lower, valueToAdd);    // 11,18,22,33
```

[1] While mostly your search ranges will be fully sorted, technically it suffices that it is partitioned on the value you are searching (partitioning is explained later).

The next example uses `equal_range()` to find the range of values equal to 2. It returns a `pair` of iterators. The first one points to the first element equal to 2, and the second points to the element after the last 2:

```
std::vector vec{ 1,2,2,3,4 };
auto result = std::equal_range(cbegin(vec), cend(vec), 2);
vec.erase(result.first, result.second);    // 1,3,4
```

Subsequence Search

Several subsequence search algorithms accept an optional binary predicate that is used to decide about equality of elements:

```
FwIt1 search(FwIt1 first1, FwIt1 last1,
             FwIt2 first2, FwIt2 last2[, BinPred predicate])
FwIt1 find_end(FwIt1 first1, FwIt1 last1,
               FwIt2 first2, FwIt2 last2[, BinPred predicate])
```

Returns an iterator to the beginning of the first (`search()`) or last (`find_end()`) subsequence in `[first1, last1)` that is equal to the range `[first2, last2)`. Returns `first1`/`last1` if the second range is empty, or `last1` if no equal subsequence is found.

```
FwIt search_n(FwIt first, FwIt last, Size count,
              const T& value[, BinPred predicate])
```

Returns an iterator to the beginning of the first subsequence in `[first, last)` that consists of `value` repeated `count` times. Returns `first` if `count` is zero, or `last` if no suitable subsequence is found.

```
FwIt search(FwIt first, FwIt last, const Searcher& searcher)
```

Returns an iterator to the beginning of the first subsequence in `[first, last)` that equals the subsequence specified in the constructor of `searcher` (see the next "Example" subsection). Returns `first` when searching the empty subsequence, and `last` if the subsequence is not found. The `<functional>` header provides the following searcher types:

- `std::default_searcher`: Naïve linear algorithm (same as `std::search()` when not given a searcher functor)
- `std::boyer_moore_searcher`: Uses the Boyer-Moore string searching algorithm, building a finite state machine upon construction to achieve sublinear performance
- `std::boyer_moore_horspool_searcher`: Uses the Boyer-Moore-Horspool string searching algorithm. Generally slower than the Boyer-Moore searcher, but less memory consuming as well

You can pass a custom binary equality predicate when constructing the searcher. For the two Boyer-Moore-based searcher types, you then likely need to provide a compatible `std::hash` functor as well (see Chapter 3)—that is, one that evaluates to the same hash for any two equal elements.

Example

The Boyer-Moore algorithms are mostly used to efficiently search for strings in larger texts. The following example requires the `<string>` and `<functional>` headers:

```
std::string needle = "hat";
std::string stack = "Burn the haystack. What's left is the needle.";
const std::boyer_moore_searcher searcher(begin(needle), end(needle));
const auto found = std::search(begin(stack), end(stack), searcher);
std::cout << "Found at position " << std::distance(begin(stack), found);
```

If multiple texts need to be searched for the same subsequence (as often occurs when searching in files, for instance), you should reuse the same searcher object.

Sequence Comparison

All the sequence comparison algorithms accept an optional binary predicate that is used to decide about equality of elements:

```
bool equal(InIt1 first1, InIt1 last1, InIt2 first2[, BinPred predicate])
```

Let n = (`last1 - first1`), then returns `true` if all elements in the ranges [`first1`, `last1`) and [`first2`, `first2` + n) pairwise match. The second range must have at least n elements. The four-argument version discussed shortly is therefore preferred to avoid out-of-bounds accesses.

```
pair<InIt1, InIt2> mismatch(InIt1 first1, InIt1 last1,
                            InIt2 first2[, BinPred predicate])
```

Let n = (`last1 - first1`), then returns a `pair` of iterators pointing to the first elements in the ranges [`first1`, `last1`) and [`first2`, `first2` + n) that do not pairwise match. The second range must have at least n elements. The four-argument version discussed next is therefore preferred to avoid out-of-bounds accesses.

```
bool equal(InIt1 first1, InIt1 last1,
           InIt2 first2, InIt2 last2[, BinPred predicate])
pair<InIt1, InIt2> mismatch(InIt1 first1, InIt1 last1,
                            InIt2 first2, InIt2 last2[, BinPred predicate])
```

Safer versions of the earlier three-argument versions that also know the length of the second range. For `equal()` to be `true`, both ranges have to be equally long. For `mismatch()`, if no mismatching pair is found before reaching either `last1` or `last2`, a pair (`first1 + m`, `first2 + m`) is returned with `m = min(last1 - first1, last2 - first2)`.

Generating Sequences

```
void fill(FwIt first, FwIt last, const T& value)
OutIt fill_n(OutIt first, Size count, const T& value)
```

Assigns `value` to all the elements in the range [`first`, `last`) or [`first`, `first + count`). Nothing happens if `count` is negative. The range for `fill_n()` must be big enough to accommodate `count` elements. `fill_n()` returns (`first + count`), or `first` if `count` is negative.

[Alternatives: `array::fill()`.]

```
void generate(FwIt first, FwIt last, Generator gen)
OutIt generate_n(OutIt first, Size count, Generator gen)
```

The generator is a function without any arguments returning a value. It is called to calculate a value for each element in the range [`first`, `last`) or [`first`, `first + count`). Nothing happens if `count` is negative. The range for `generate_n()` must be big enough to accommodate `count` elements. `generate_n()` returns (`first + count`), or `first` if `count` is negative.

```
void iota(FwIt first, FwIt last, T value)
```

This algorithm is defined in the `<numeric>` header. Each element in the range [`first`, `last`) is set to `value`, after which `value` is incremented, so

```
*first = value++
*(first + 1) = value++
*(first + 2) = value++
...
```

Example

The following example demonstrates `generate()` and `iota()`:

```
std::vector<int> vec(6);                        // 0,0,0,0,0,0
int value = 11;
std::generate(begin(vec), begin(vec) + 3,
   [&value] { value *= 2; return value; });     // 22,44,88,0,0,0
std::iota(begin(vec) + 3, end(vec), 2);         // 22,44,88,2,3,4
```

Copy, Move, Swap

`OutIt copy(InIt first, InIt last, OutIt targetFirst)`
`OutIt copy_if(InIt first, InIt last, OutIt targetFirst, UnaPred predicate)`

Copies either all the elements (`copy()`) or only those that satisfy a unary `predicate` (`copy_if()`), from the range `[first, last)` to a range starting at `targetFirst`. For `copy()`, `targetFirst` is not allowed to be in `[first, last)`: if this is the case, `copy_backward()` may be an option. For `copy_if()`, the ranges are not allowed to overlap. For both algorithms, the target range must be big enough to accommodate the copied elements. Returns the end iterator of the resulting range.

`BidIt2 copy_backward(BidIt1 first, BidIt1 last, BidIt2 targetLast)`

Copies all the elements in the range `[first, last)` to a range ending at `targetLast`, which is not in the range `[first, last)`. The target range must be big enough to accommodate the copied elements. Copying is done backward, starting with copying element `(last-1)` to `(targetLast-1)` and going back to `first`. Returns an iterator to the beginning of the target range, so `(targetLast - (last - first))`.

`OutIt copy_n(InIt start, Size count, OutIt target)`

Copies `count` elements starting at `start` to a range starting at `target`. The target range must be big enough to accommodate the elements. Returns the target end iterator, so `(target + count)`.

`OutIt move(InIt first, InIt last, OutIt targetFirst)`
`BidIt2 move_backward(BidIt1 first, BidIt1 last, BidIt2 targetLast)`

Similar to `copy()` and `copy_backward()` but moves the elements instead of copying them.

`FwIt2 swap_ranges(FwIt1 first1, FwIt1 last1, FwIt2 first2)`

Swaps the elements in the range `[first1, last1)` with the elements in the range `[first2, first2 + (last1 - first1))`. Both ranges are not allowed to overlap, and the second range must be at least as big as the first. Returns an iterator one past the last swapped element in the second range.

`void iter_swap(FwIt1 x, FwIt2 y)`

Swaps the element pointed to by `x` with the element pointed to by `y`, so `swap(*x, *y)`.

Example

The following example copies all even numbers from one `vector` to another. To make sure the target range is always sufficiently large, we first make the target `vector` larger than it probably should be, and then `erase()` any excess elements:

```
std::vector<int> numbers{ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 };
std::vector<int> evens(numbers.size());
auto end = std::copy_if(cbegin(numbers), cend(numbers), begin(evens),
                                        [](int x) { return x % 2 == 0; });
evens.erase(end, evens.end());     // 2 8 34 144
```

Preallocating and then erasing like this is rather cumbersome. At the end of this chapter, we will show you how to use `copy_if()` more conveniently in combination with iterator adaptors.

Removing and Replacing

`FwIt remove(FwIt first, FwIt last, const T& value)`
`FwIt remove_if(FwIt first, FwIt last, UnaPred predicate)`

Moves all elements in the range `[first, last)` that are not equal to `value`, or do not satisfy a unary `predicate`, toward the beginning of the range, after which `[first, result)` contains all the elements to keep. The `result` iterator, pointing to one past the last element to keep, is returned. The algorithms are stable, which means the retained elements maintain their relative order. The elements in `[result, last)` should not be used because they could be in an unspecified state due to moves. Usually these algorithms are followed by a call to `erase()`. This is known as the *remove-erase idiom* and is discussed in Chapter 3.

[Alternatives: `list` and `forward_list` have `remove()` and `remove_if()` members.]

```
FwIt unique(FwIt first, FwIt last[, BinPred predicate])
```

Removes all but one element from consecutive equal elements in the range [first, last). If a binary predicate is given, it is used to decide about equality of elements. Otherwise equivalent to remove(), including the fact that it should normally be followed by an erase(). A typical use of unique() is shown in the next "Example" subsection.

[Alternatives: list::unique() and forward_list::unique().]

```
void replace(FwIt first, FwIt last, const T& oldVal, const T& newVal)
void replace_if(FwIt first, FwIt last, UnaPred predicate, const T& newVal)
```

Replaces with newVal all elements in the range [first, last) equal to oldVal, or satisfying a unary predicate.

```
OutIt remove_copy(InIt first, InIt last, OutIt target, const T& value)
OutIt remove_copy_if(InIt first, InIt last, OutIt target, UnaPred pred)
OutIt unique_copy(InIt first, InIt last, OutIt target[, BinPred predicate])
OutIt replace_copy(InIt first, InIt last, OutIt target,
                   const T& oldVal, const T& newVal)
OutIt replace_copy_if(InIt first, InIt last, OutIt target,
                      UnaPred predicate, const T& newVal)
```

Similar to the previous algorithms, but copies the results to a range starting at target. The target range must be big enough to accommodate the copied elements. The input and target ranges are not allowed to overlap. Returns an iterator pointing to one past the last element that was copied to the target range.

Example

The following example demonstrates the use of unique() and the remove-erase idiom to filter out all consecutive equal elements from a vector:

```
std::vector v{ 3,4,4,4,5,6,4,5,5,7 };
auto result = std::unique(begin(v), end(v));
                                // possible outcome: 3,4,5,6,4,5,7,5,5,7
v.erase(result,end(v));         // final outcome:    3,4,5,6,4,5,7
```

Reversing and Rotating

```
void reverse(BidIt first, BidIt last)
```

Reverses the elements in the range [first, last).

[Alternatives: list::reverse() and forward_list::reverse().]

```
FwIt rotate(FwIt first, FwIt middle, FwIt last)
```

Rotates the elements in the range [first, last) to the left in such a way that the element pointed to by middle becomes the first element in the range and the element pointed to by (middle - 1) becomes the last element in the range (see the next "Example" subsection). Returns (first + (last - middle)).

```
OutIt reverse_copy(BidIt first, BidIt last, OutIt target)
OutIt rotate_copy(FwIt first, FwIt middle, FwIt last, OutIt target)
```

Similar to reverse() and rotate(), but copies the results to a range starting at target. The target range must be big enough to accommodate the copied elements. The input and target ranges are not allowed to overlap. Returns an iterator pointing to one past the last element copied to the target range.

Example

The next code snippet rotates the elements in the vector. The result is 5,6,1,2,3,4:

```
std::vector vec{ 1,2,3,4,5,6 };
std::rotate(begin(vec), begin(vec) + 4, end(vec));
```

Partitioning

```
bool is_partitioned(InIt first, InIt last, UnaPred predicate)
```

Returns true if the elements in the range [first, last) are partitioned such that all elements satisfying a unary predicate are before all elements that do not satisfy the predicate. Also returns true if the range is empty.

```
FwIt partition(FwIt first, FwIt last, UnaPred predicate)
BidIt stable_partition(BidIt first, BidIt last, UnaPred predicate)
```

Partitions the range [first, last) such that all elements satisfying a unary predicate are before all elements that do not satisfy the predicate. Returns an iterator to the first element that does not satisfy the predicate. stable_partition() maintains the relative order of elements in both partitions.

```
pair<OutIt1, OutIt2> partition_copy(InIt first, InIt last,
    OutIt1 outTrue, OutIt2 outFalse, UnaPred predicate)
```

Partitions the range [first, last) by copying all elements that satisfy or do not satisfy a unary predicate to an output range starting at outTrue or outFalse, respectively. Both output ranges must be big enough to accommodate the copied elements. The input and output ranges are not allowed to overlap. Returns a pair containing the end iterator of the two output ranges.

```
FwIt partition_point(FwIt first, FwIt last, UnaPred predicate)
```

Requires the range [first, last) to be partitioned based on a unary predicate. Returns an iterator to the first element of the second partition: that is, the first element that does not satisfy the predicate.

Sorting

```
void sort(RanIt first, RanIt last[, Compare comp])
void stable_sort(RanIt first, RanIt last[, Compare comp])
```

Sorts the elements in the range [first, last). The stable version maintains the order of equal elements.

[Alternatives: `list::sort()` and `forward_list::sort()`.]

```
void partial_sort(RanIt first, RanIt middle, RanIt last[, Compare comp])
```

The (middle - first) smallest elements from the range [first, last) are sorted and moved to the range [first, middle). The unsorted elements are moved to the range [middle, last) in an unspecified order.

```
RanIt partial_sort_copy(InIt first, InIt last,
  RanIt targetFirst, RanIt targetLast[, Compare comp])
```

min(last - first, targetLast - targetFirst) elements from the range [first, last) are sorted and copied to the target range. Returns min(targetLast, targetFirst + (last - first)).

```
bool is_sorted(FwIt first, FwIt last[, Compare comp])
```

Returns true if the range [first, last) is a sorted sequence.

```
FwIt is_sorted_until(FwIt first, FwIt last[, Compare comp])
```

Returns the last iterator, iter, such that [first, iter) is a sorted sequence.

```
bool lexicographical_compare(InIt1 first1, InIt1 last1,
  InIt2 first2, InIt2 last2[, Compare comp])
```

Returns whether the elements in the range [first1, last1) are lexicographically less than the elements in the range [first2, last2).

```
void nth_element(RanIt first, RanIt nth, RanIt last[, Compare comp])
```

The elements in the range [first, last) are moved in such a way that the given iterator nth, after rearranging, points to the element that would be in that position if the whole range were sorted. The entire range does not actually get sorted. It is, however, (nonstably) partitioned on the element nth points to.

Example

The `partial_sort()` and `partial_sort_copy()` algorithms can be used to find the *n* biggest, smallest, worst, best, ... elements in a sequence. This is faster than sorting the entire sequence. For example:

```
std::vector<int> vec{ 9,2,4,7,3,6,1 };
std::vector<int> threeSmallestElements(3);
std::partial_sort_copy(begin(vec), end(vec),
   begin(threeSmallestElements), end(threeSmallestElements));
```

`nth_element()` is a so-called *selection algorithm* to find the nth smallest number in a sequence and has on average a linear complexity. It can, for example, be used to calculate the median value of a sequence with an odd number of elements:

```
std::vector vec{ 9,2,4,7,3,6,1 };
auto middle = begin(vec) + vec.size() / 2;
std::nth_element(begin(vec), middle, end(vec));
int median = *middle;    // 4
```

`nth_element()` is not only a selection algorithm, though, but also a partition algorithm. If sorting is not a requirement, it can be used to retrieve the n smallest elements of a series even faster than `partial_sort()`.

```
std::vector vec{ 9,2,4,7,3,6,1 };
std::nth_element(begin(vec), begin(vec) + 3, end(vec));
std::vector threeSmallest(begin(vec), begin(vec) + 3); // Possible: 2,1,3
```

Sampling and Shuffling

The algorithms we discuss in this section draw their randomness from one of the uniform random number generators explained in Chapter 1.

C++17

```
void sample(InIt first, InIt last, RanIt out, Size n, UniformRanGen gen)
void sample(FwIt first, FwIt last, OutIt out, Size n, UniformRanGen gen)
```

Copies `min(last - first, n)` randomly selected elements from the range `[first, last)` to the range starting at `out`. Each element has equal probability of being selected, and no element is selected more than once. This algorithm is stable (in the sense that the elements in the output appear in the same order as in the input range) if the input iterators are forward iterators.

```
void shuffle(RanIt first, RanIt last, UniformRanGen generator)
```

Shuffles the elements in the range `[first, last)`.

Example

The following example first randomly samples six numbers from a `vector` and then shuffles these. See Chapter 1 for more information on the random number generation library. The code snippet additionally needs `<random>` and `<ctime>`:

```
std::random_device seeder;
const auto seed = seeder.entropy() ? seeder() : std::time(nullptr);
std::default_random_engine gen(
   static_cast<std::default_random_engine::result_type>(seed));
std::vector<int> in{ 1,2,3,4,5,6,7,8,9,10 };
std::vector<int> out(6);
std::sample(begin(in), end(in), begin(out), size(out), gen);
                                          // Possible result: 1 3 4 7 9 10
std::shuffle(begin(out), end(out), gen);  // Possible result: 3 10 9 7 4 1
```

Operations on Sorted Ranges

All the following operations require that the input ranges are sorted. If this precondition is not met, the algorithms' behavior is undefined. Note that the difference between `merge()` and `set_union()` lies in how they behave if equivalent elements appear in both their input ranges:

```
bool includes(InIt1 first1, InIt1 last1,
              InIt2 first2, InIt2 last2[, Compare comp])
```

Returns `true` if all elements in the sorted range `[first2, last2)` are in the sorted range `[first1, last1)` or if the former is empty, or `false` otherwise.

```
OutIt merge(InIt1 first1, InIt1 last1,
            InIt2 first2, InIt2 last2, OutIt target[, Compare comp])
```

Merges all the elements from the sorted ranges `[first1, last1)` and `[first2, last2)` to a range starting at `target` in such a way that the target range is sorted as well. The target range must be big enough to accommodate all elements. The input ranges are not allowed to overlap with the target range. Returns the end iterator of the target range. The algorithm is stable; that is, the order of equal elements is maintained. For elements that appear in both ranges, the elements of the first range are copied to the output before those of the second.

[Alternatives: `list`, `forward_list`, as well as all associative containers offer `merge()` members.]

```
void inplace_merge(BidIt first, BidIt middle, BidIt last[, Compare comp])
```

Merges the sorted ranges `[first, middle)` and `[middle, last)` into one sorted sequence stored in the range `[first, last)`. The algorithm is stable, so the order of equal elements is maintained.

```
OutIt set_union(InIt1 first1, InIt1 last1,
                InIt2 first2, InIt2 last2, OutIt target[, Compare comp])
OutIt set_intersection(InIt1 first1, InIt1 last1,
                InIt2 first2, InIt2 last2, OutIt target[, Compare comp])
OutIt set_difference(InIt1 first1, InIt1 last1,
                InIt2 first2, InIt2 last2, OutIt target[, Compare comp])
OutIt set_symmetric_difference(InIt1 first1, InIt1 last1,
                InIt2 first2, InIt2 last2, OutIt target[, Compare comp])
```

Performs set operations (see the following list) on two sorted ranges `[first1, last1)` and `[first2, last2)` and stores the results in a range starting at `target`. The elements copied to the target are sorted. The target range must be big enough to hold all output elements. The input and output ranges are not allowed to overlap. Returns the end iterator of the constructed target range. If an element appears *m* times in the first input range, and *n* times in the second input range, then

- *Union*: The element is copied m times from the first range, and then max(0, *n*-m) times from the second. The element therefore appears max(*m*, *n*) times in the output.
- *Intersection*: The element is copied min(*m*, *n*) times from the first range.
- *Difference*: The element is copied max(0, m-n) times from the first range.
- *Symmetric difference*: If $m > n$, then the last $m-n$ equivalent elements are copied from the first input range; if $n > m$, then the $n-m$ last ones are copied from the second. The element therefore appears $|m-n|$ times in the output.

Permutation

```
bool is_permutation(FwIt1 first1, FwIt1 last1,
                    FwIt2 first2[, BinPred predicate])
bool is_permutation(FwIt1 first1, FwIt1 last1,
                    FwIt2 first2, FwIt2 last2[, BinPred predicate])
```

Returns `true` if the second range is a permutation of the first one. For the three-argument versions, the second range is defined as `[first2, first2 + (last1 - first1))`, and this range must be at least as large as the first. The four-argument versions are therefore preferred to safeguard against out-of-bounds accesses (they return `false` if the ranges have different lengths). If a binary `predicate` is given, it is used to decide about equality of elements between the two ranges.

```
bool next_permutation(BidIt first, BidIt last[, Compare comp])
bool prev_permutation(BidIt first, BidIt last[, Compare comp])
```

Transforms the elements in the range [`first`, `last`) into the lexicographically next/previous permutation. Returns `true` if such a next/previous permutation exists, otherwise returns `false` and transforms the elements in the lexicographically smallest/largest permutation possible.

Heaps

In this context, the term *heap* does not refer to the dynamic memory pool of the C++ runtime. In computer science, heaps are also a family of fundamental tree-based data structures (well-known variants include binary, binomial, and Fibonacci heaps). These data structures are key building blocks in the efficient implementation of various graph and sorting algorithms (classic examples include Prim's algorithm, Dijkstra's algorithm, and heapsort). It is also a common implementation strategy for a priority queue: in fact, the C++ `priority_queue` container adaptor discussed in the previous chapter could easily be implemented using the heap algorithms defined next.

For the following C++ algorithms, the heap's tree is flattened into a contiguous sequence of elements that is ordered in a particular fashion. Although the exact ordering is implementation specific, it must satisfy the following key properties: its first element is always the largest (or one of the largest elements), and both removing this largest element (`pop_heap()`) and adding new elements (`push_heap()`) can be done repeatedly in logarithmic time.

```
void make_heap(RanIt first, RanIt last[, Compare comp])
```

Turns the range [`first`, `last`) into a heap (in linear time).

```
void push_heap(RanIt first, RanIt last[, Compare comp])
```

The last element of the range [`first`, `last`) is moved to the correct position such that it becomes a heap. The range [`first`, `last - 1`) is required to be a heap prior to calling `push_heap()`.

```
void pop_heap(RanIt first, RanIt last[, Compare comp])
```

Removes the greatest element from the heap [`first`, `last`) by swapping `*first` with `*(last - 1)` and making sure the new range [`first`, `last - 1`) remains a heap.

```
void sort_heap(RanIt first, RanIt last[, Compare comp])
```

Sorts all the elements in the range [`first`, `last`). The range is required to be a heap prior to calling `sort_heap()`.

`bool is_heap(RanIt first, RanIt last[, Compare comp])`

Returns `true` if the range `[first, last)` represents a heap.

`RanIt is_heap_until(RanIt first, RanIt last[, Compare comp])`

Returns the last iterator, `iter`, such that `[first, iter)` represents a heap.

Numeric Algorithms <numeric>

Reductions

Reduce functions (which are also called *accumulate*, *aggregate*, *compress*, *fold*, or *inject* functions) take a range of elements and repeatedly combine two elements until only one value is left. The default combination operator is typically summation.

`T accumulate(InIt first, InIt last, T startValue[, BinOp op])`

Returns `result`, which is calculated by initializing `result` with `startValue` and then executing `result = result + element` or `result = op(result, element)` for each `element` in the range `[first, last)`. `accumulate()` thus performs what is known as a *left fold*. To perform a *right fold* instead, you need to reverse both the input range (or use reverse iterators) and the order of the arguments to the binary operator.

C++17

```
T reduce(InIt first, InIt last)
T reduce(InIt first, InIt last, T startValue[, BinOp op])
```

Same as `accumulate()`, except that `std::plus<>` is used by default instead of `operator+` and that the elements may be grouped and rearranged arbitrarily while accumulating. The latter facilitates the efficient parallelization of this algorithm (as discussed later in this chapter), but also implies that `op` should be both commutative and associative.

C++17

`T transform_reduce(Init first, Init last, T start, BinOp op1, UnaOp op2)`

Same as `reduce()`, except that a given unary operator is applied to each element of the input range first, before they are combined with any other values. The unary operation is not applied to `start`.

`transform_reduce()` can also perform reductions of two ranges. These overloads are discussed in the next section.

Example

The following code snippet uses the `reduce()` algorithm to calculate the sum of all elements in a sequence:

```
std::vector vec{ 4,2,5,1,3,6 };
int sum = std::reduce(begin(vec), end(vec));    // 21
```

Inner Products

An *inner product* (also: *dot product* or *scalar product*) reduces not one but two ranges to a single value by first combining corresponding elements of two ranges and then reducing these combined values again to a single value:

```
T inner_product(InIt1 first1, InIt1 last1, InIt2 first2,
                T startValue[, BinOp1 op1, BinOp2 op2])
```

Returns `result`, which is calculated by starting with `result` equal to `startValue` and then executing `result = result + (el1 * el2)` or `result = op1(result, op2(el1, el2))` for each `el1` from the range `[first1, last1)` and each `el2` from the range `[first2, first2 + (last1 - first1))` in order. The second range must be at least as big as the first.

C++17

```
T transform_reduce(InIt1 first1, InIt1 last1, InIt2 first2,
                   T startValue[, BinOp op1, BinOp2 op2])
```

Same as `inner_product()`, except that `std::plus<>` and `multiplies<>` are used by default and that corresponding elements from both ranges may be grouped and rearranged arbitrarily while accumulating. The latter facilitates the efficient parallelization of this algorithm (as discussed later in this chapter), but does imply that `transform_reduce()` should only be applied with commutative and associative `op1` operators.

Example

The dot product of two vectors is a very common operation in 3D geometry:

```
double v1[] = { 0,1,2 };
double v2[] = { 1,0,2 };
double dot = std::inner_product(std::begin(v1), std::end(v1),
   std::begin(v2), 0.0);    // 0*1 + 1*0 + 2*2 = 4.0
```

Prefix Sums

The *prefix sum* (also: *cumulative sum*, *partial sum*, or *scan*) of a sequence of numbers is a second sequence that consists of sums of these numbers, where each next sum adds one more number from the input sequence:

```
OutIt partial_sum(InIt first, InIt last, OutIt target[, BinOp op])
```

Calculates partial sums of increasing subranges from [`first`, `last`) and writes the results to a range starting at `target`. With the default operator, +, the result is as if calculated as follows:

```
*(target) = *first
*(target + 1) = *first + *(first + 1)
*(target + 2) = *first + *(first + 1) + *(first + 2)
...
```

Returns the end iterator of the target range, so (`target` + (`last` - `first`)). The target range must be big enough to accommodate the results. The calculations can be done in place by specifying `target` equal to `first`.

C++17

```
OutIt inclusive_scan(InIt first, InIt last, OutIt target[, BinOp op])
OutIt inclusive_scan(InIt first, InIt last, OutIt target, BinOp op,
                                                          T startValue)
```

Same as `partial_sum()`, except that `std::plus<>` is used by default and that the summation may be performed in any order. The latter facilitates the efficient parallelization of this algorithm (as discussed later in this chapter), but also implies that `op` should always be commutative and associative. Each partial sum is initialized with `startValue`, if such a value is provided.

C++17

```
OutIt exclusive_scan(InIt first, InIt last, OutIt target,
                     T startValue[, BinOp op])
```

Same as `inclusive_scan()`, except that `*(first` + *i*) is not included in the *i*-th partial sum (hence the term 'exclusive'). In other words, the output could be computed as follows:

```
*(target) = startValue
*(target + 1) = startValue + *first
*(target + 2) = startValue + *first + *(first + 1)
...
```

(Of course, like with `inclusive_scan()`, computations may be rearranged freely, and `std::plus<>` is used by default rather than `operator+`.)

C++17
```
OutIt transform_inclusive_scan(InIt first, InIt last, OutIt target,
                               BinOp op1, UnOp op2[, T startValue])
OutIt transform_exclusive_scan(InIt first, InIt last, OutIt target,
                               T startValue, BinOp op1, UnOp op2)
```

Same as `inclusive_scan()` / `exclusive_scan()`, except that a given unary operation is applied to each element of the input range before values are added together (the unary operation is not applied to `startValue`).

Element Differences

```
OutIt adjacent_difference(InIt first, InIt last, OutIt target[, BinOp op])
```

Calculates differences of adjacent elements in the range `[first, last)` and writes the results to a range starting at `target`. For the default operator, `-`, the result is calculated as follows:

```
*(target) = *first
*(target + 1) = *(first + 1) - *first
*(target + 2) = *(first + 2) - *(first + 1)
...
```

Returns the end iterator of the target range, so `(target + (last - first))`.
The target range must be big enough to accommodate the results.
The calculations can be done in place by specifying `target` equal to `first`.

Algorithms for Uninitialized Memory <memory>

Because they never invoke any destructor or assignment operator to deinitialize any preexisting objects in their target ranges, most algorithms in this section require these target ranges to consist of uninitialized memory. You can obtain blocks of uninitialized dynamic memory either from a C++ allocator (see Chapter 3) or from more low-level, C-style allocation facilities such as `malloc()` and `aligned_alloc()` (both defined in `<cstdlib>`, but not further explained in this book).

```
void uninitialized_default_construct(FwIt first, FwIt last) C++17
FwIt uninitialized_default_construct_n(FwIt first, Size count)
void uninitialized_value_construct(FwIt first, FwIt last)
FwIt uninitialized_value_construct_n(FwIt first, Size count)
```

Initializes a sequence of values as if by placement new with either default (`new (`*`address`*`) T`) or value (`new (`*`address`*`) T()`) construction. The variants that accept an iterator and a `count` return an iterator pointing one past the last constructed element.

```
FwIt uninitialized_copy(InIt first, InIt last, FwIt result)
FwIt uninitialized_copy_n(InIt first, Size count, FwIt result)
FwIt uninitialized_move(InIt first, InIt last, FwIt result) C++17
pair<InIt, FwIt> uninitialized_move_n(InIt first, Size count, FwIt result)
```

Copies/moves a range of values to the range starting at `result` as if by placement new with copy/move construction. This is unlike the regular `copy()` / `move()` algorithms which use assignments (or equivalent). The first three algorithms return an iterator that points into the *target* range, one past the last element that was copied/moved. In the `pair` of iterators that `uninitialized_move_n()` returns, the second element equals that same iterator as well. The first iterator of that `pair` points into the *source* range, one past the last element that was moved (without raising an exception).

```
void uninitialized_fill(FwIt first, FwIt last, const T& value)
FwIt uninitialized_fill_n(FwIt first, Size count, const T& value)
```

Fills the entire given range with copies of `value` as if by placement new with copy construction. This is unlike the plain `fill()` algorithms, which instead use copy assignment (or equivalent). The second algorithm returns an iterator pointing one past the last element that was initialized.

```
void destroy_at(T* location) C++17
void destroy(FwIt first, FwIt last)
FwIt destroy_n(FwIt first, Size count)
```

Invokes the destructor for the element at the given location, or for all elements in the given range, without deallocating any memory. The result is a block of uninitialized memory, which may be used as the target for any of the other algorithms in this section. `destroy_n()` returns an iterator pointing one past the last element that was destroyed.

Parallel Algorithms C++17 <execution>

Starting with C++17, nearly all algorithms defined in the `<algorithm>`, `<numeric>`, and `<memory>` headers can be executed in parallel simply by passing a so-called *execution policy object* as the first argument. In the following snippet, for instance, we use `std::sort()` to sort the `largeVector` container in parallel:

```
using namespace std::execution;
std::sort(par, begin(largeVector), end(largeVector));
```

By passing `std::execution::par`, a global constant of type `std::execution::parallel_policy`, we signal the library that it can use any number of threads (the calling thread may be one of them), on any number of processing units.

The `<execution>` header defines the following three execution policy objects and types, all in the `std::execution` namespace:

Object	Type	Summary
`seq`	`sequenced_policy`	Execution may not be parallelized.
`par`	`parallel_policy`	Execution may be *parallelized*. Algorithm function invocations do not interleave on the same thread.
`par_unseq`	`parallel_unsequenced_policy`	Execution may be *parallelized*. To facilitate *vectorization*, algorithm function invocations may be interleaved on the same thread.

Implementations are free to define additional execution policy types. The only requirement is that `std::is_execution_policy_v<PolicyType>` evaluates to `true`.

■ **Tip** Use parallel execution when operating on large data ranges, and/or when a significant amount of work needs to be performed per element. The latter mostly occurs with algorithms such as `std::for_each` or any of the reduction algorithms. Either way, best is you always profile to verify that parallel execution is effectively more efficient.

■ **Caution** If a user-provided function exits with an uncaught exception during the parallel execution of an algorithm with any of the three standard execution policies (so even if you use a policy object of type `sequenced_policy`), `std::terminate()` is called, which by default terminates the process (see Chapter 8).

■ **Note** At the time of writing, not all compilers (fully) support parallel algorithms yet: of the compilers we verified, only GCC 9.1 recently added full support (requires `-ltbb` to link with Intel TBB 2018 or later), Visual Studio 2019 only has partial support (all overloads are added, but not all algorithms are effectively parallelized), and Clang 8 has no support for parallel algorithms at all yet. Consult your compiler's documentation for more details. If your compiler does not support parallel algorithms yet, you may consider third-party implementations, such as Intel Parallel STL or HPX.

Parallel Execution

With `std::execution::par`, it is your responsibility to ensure that parallel function invocations do not result in data races or deadlocks. Consider the following example:

```
std::for_each(par, begin(input), end(input),
  [&vector](const auto& elem) { vector.push_back(process(elem)); });
```

If `vector` is a regular `std::vector`, parallel execution clearly results in data races: concurrently writing to shared data is not safe without synchronization. To fix this example, you could acquire a shared mutex once `process()` is done, right before calling `push_back()`. Provided, of course, it is safe to run `process()` concurrently. If it is not, however, there really is no point in running this loop in parallel. We refer to Chapter 7 for more information on concurrency issues and thread synchronization.

Parallel Unsequenced Execution

The term *unsequenced* refers to the fact that when you pass `par_unseq` to an algorithm, instructions of multiple algorithm function invocations are allowed to be interleaved even on a single thread of execution. Allowing instructions of multiple invocations to be reordered more freely facilitates *vectorization*. Vectorization is the replacement of multiple hardware instructions with a single *vector instruction*—a special hardware instruction that performs the same (numeric) operation on multiple values at once. Given enough data, these vector instructions may even be run on multiple threads at once, resulting in even more speedup.

Unsequenced execution, however, does come at a cost: it limits the functionality you may use inside your algorithm functions. In technical terms, you are not allowed to invoke any Standard Library function that is specified to *synchronize* with another function, or for which any other function is specified to synchronize with it. Memory allocation and deallocation functions are exempted from this restriction. Notable examples of synchronizing functions include the `lock()` and `unlock()` functions of mutexes and functions of `std::atomics` that require locks. We again refer to Chapter 7 for more details on synchronization.

Iterator Adaptors `<iterator>`

The Standard Library provides five iterator adaptors. These first two are created from a given iterator and are typically used for *input* ranges of algorithms:

- `move_iterator`: Applies `std::move()` (or equivalent) to the result of dereferencing a given iterator
- `reverse_iterator`: Reverses the order of a given iterator

The other three iterator adaptors are created from a given container (not an iterator) and generally act as special *output* iterators to algorithms:

- `back_insert_iterator`: Adds elements to a given container using `push_back()`
- `front_insert_iterator`: Adds elements to a given container using `push_front()`
- `insert_iterator`: Adds elements to a given container using `insert()`

To ease the creation of these last three adaptors, the Standard Library offers these factory functions: `std::back_inserter()`, `front_inserter()`, and `inserter()`. Here is an example:

```
std::vector nums{ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 };
std::vector<int> evens;
auto is_even = [](int x) { return x % 2 == 0; };
std::copy_if(cbegin(nums), cend(nums), back_inserter(evens), is_even);
```

Notice also how argument-dependent lookup (ADL) allows you to omit '`std::`' from '`std::back_inserter()`' when targeting a standard container.

You often use a `std::back_insert_iterator` like in our previous example in combination with algorithms such as `std::copy_if()`, `unique_copy()`, and so on, to copy an unknown amount of elements into a sequential container. Even if the size of the target range is known, using a `back_insert_iterator` is often more convenient than first resizing the target container. When inserting larger amounts of data, you could still consider using `reserve()` first to increase the capacity.

If the target container is an associative container, you use an `insert_iterator` instead. Next to the container, you then also pass the insertion position:

```
std::set<int> odds;
std::remove_copy_if(cbegin(nums), cend(nums),
                    inserter(odds, begin(odds)), is_even);
```

■ **Note** Factory functions also exist for the other two iterator adaptors: `std::make_reverse_iterator()` and `make_move_iterator()`. But these factory functions have lost their appeal now that C++17 has introduced class template argument deduction. Consider `std::reverse_iterator(iter)`—a valid constructor invocation since C++17. Since this is already shorter than `std::make_reverse_iterator(iter)`, there's little reason to still use the latter.

■ **Note** The three inserting iterator adaptors cannot be used as output iterator for parallel algorithms. In technical terms, the reason is that these output iterators are not forward iterators, a requirement that parallelized algorithm functions typically impose for their output iterators. The practical reason though is clear enough: inserting elements concurrently into a container is simply not safe.

CHAPTER 5

Input/Output

In C++, input and output (*I/O* for short) mostly happens through an abstraction known as *streams*. Streams allow you to perform I/O operations without knowing the details of their target or source. Be it your command-line interface, a string, or a file—streams offer an easy, uniform interface to communicate with any of these.

In this chapter, we first cover the different C++ stream libraries. Next, we discuss the C++17 `<filesystem>` library, which allows you to inspect and manipulate the files that are present on your file system. We conclude the chapter with a brief introduction of some useful C-style I/O functions.

Input/Output with Streams

The stream classes provided by the Standard Library are organized in a hierarchy and a set of headers, as shown in Figure 5-1.

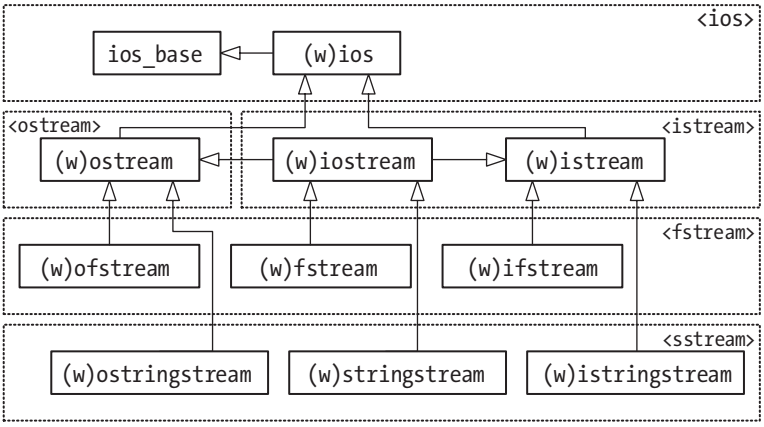


Figure 5-1. *The hierarchy of stream-related classes*

P. Van Weert and M. Gregoire, *C++17 Standard Library Quick Reference*,
https://doi.org/10.1007/978-1-4842-4923-9_5

More accurately, the library defines templates called `basic_ios`, `basic_ostream`, `basic_istringstream`, and so on, all parameterized on a character type. All classes in the hierarchy, except `ios_base`, are type aliases for these templates with either `char` or `wchar_t` as template type argument. For example, `ostream` is an alias for `basic_ostream<char>`, and `wifstream` is an alias for `basic_ifstream<wchar_t>`. The remainder of this chapter only uses the `char` aliases.

In addition to the headers depicted in Figure 5-1, there are also headers `<iosfwd>` and `<iostream>`:

- `<iosfwd>` simply contains forward declarations of all stream-related types.
- `<iostream>` includes `<ios>`, `<streambuf>`, `<istream>`, `<ostream>`, and `<iosfwd>` while itself adding the standard input and output streams (`w`)`cin`, (`w`)`cout`, (`w`)`cerr`, and (`w`)`clog`.

Note that, somewhat confusingly, `<iostream>` is thus not a header that defines `std::iostream` (at least not directly)—this type is defined in the `<istream>` header instead.

The library also provides the `std::basic_streambuf`, `basic_filebuf`, and `basic_stringbuf` templates and their various type aliases, plus `istreambuf_iterator` and `ostreambuf_iterator`. These are, or work on, *stream buffers* and are the basis for the implementation of other stream classes, such as `ostream`, `ifstream`, and so on. Stream buffers and iterators are discussed later in this chapter.

Helper Types `<ios>`

The following helper types are defined in `<ios>`:

Type	Description
`std::streamsize`	An alias for a signed integral type used to represent the number of characters transferred during an I/O operation or to represent the size of an I/O buffer.
`std::streamoff`	An alias for a signed integral type used to represent an offset into a stream.
`std::fpos`	A class template containing an absolute position in a stream and a conversion operator to convert it into a `streamoff`. Certain arithmetic operations are supported: a `streamoff` can be added to or subtracted from an `fpos`, resulting in an `fpos` (using `+`, `-`, `+=`, or `-=`), and two `fpos` objects can be compared (using `==` or `!=`) or subtracted, resulting in a `streamoff` (using `-`). Predefined type aliases are provided: `std::streampos` and `wstreampos` for the character types `char` and `wchar_t`.

Formatting Methods (std::ios_base) <ios>

The `ios_base` class, defined in `<ios>`, is the base class for all input and output stream classes. It keeps track of formatting options and flags to manipulate how data is read and written. The following methods are provided to manipulate text formatting:

Method	Description
`precision()` `precision(streamsize)`	Returns the precision for floating-point I/O, or changes it while returning the old one. The semantics of the precision depend on which `floatfield` formatting flag is set (see Tables 5-1 and 5-2). If either `fixed` or `scientific` is set, the precision specifies *exactly* how many digits to show *after* the decimal separator, even if this means adding trailing zeros. If neither is set, then it denotes the *maximum* number of digits to show, counting both the digits *before and after* the decimal separator (no zeros are added in this case). And if both are set, the precision is ignored.
`width()` `width(streamsize)`	Returns the width of the next field, or changes it while returning the old one. This width specifies the minimum number of characters to output with certain I/O operations. To reach this minimum, fill characters (explained later) are added. Only has an effect on the next I/O operation.
`getloc()` `imbue(locale)`	Returns the `locale` used during I/O, or changes it while returning the old one. See Chapter 6 for details on locales.
`flags()` `flags(fmtflags)`	Returns the currently set formatting flags, or replaces the current flags while returning the old ones. Table 5-1 lists all available `fmtflags` flags, which can be combined bitwise.
`setf(fmtflags)` `unsetf(fmtflags)`	Sets or unsets individual flags without touching others. The flags prior to the update are returned.
`setf(fmtflags flags,` `fmtflags mask)`	Sets `flags` while unsetting others in a group, specified as a `mask`. Table 5-2 lists the predefined masks. For example, `setf(right | fixed, adjustfield | floatfield)` sets the `right` and `fixed` flags while unsetting the `left`, `internal`, and `scientific` flags.

You can also modify flags by streaming one of the I/O manipulators discussed in the next section.

■ **Caution** Stream formatting settings are mostly *sticky* in the sense that they impact all subsequent I/O operations on the stream. The only exception is `width()`: most stream I/O operations reset the width parameter to zero (meaning "unspecified"). To read/write multiple values with the same width, `width()` (or the corresponding I/O manipulator `setw()`: see later) needs to be repeated before each I/O operation.

Table 5-1. *`std::ios_base::fmtflags` Formatting Flags Defined in `<ios>`*

Flag	Description
`boolalpha`	Use `true` and `false` instead of `1` and `0` for Boolean I/O.
`left`, `right`, `internal`	Output is `left` aligned with fill characters added to the right, or `right` aligned with padding on the left, or adjusted by padding in the middle. `internal` works for numerical and monetary values, with the designated padding point being between the value and any of its prefixes: a sign, numerical base, and/or currency symbol. Otherwise, `internal` is equivalent to `right`. The results of the different alignment options are shown in the example section.
`scientific`, `fixed`	If neither of these flags is set, use default notation for floating-point I/O (for instance, `0.0314`). Otherwise, use scientific (`3.140000e-02`) or fixed notation (`0.031400`). If both are combined, `scientific | fixed`, use hexadecimal floating-point notation (`0x1.013a92p-5`).
`dec`, `oct`, `hex`	Use a decimal, octal, or hexadecimal base for integer I/O.
`showbase`	For integer I/O, write or expect the base prefix as specified with `dec`, `oct`, or `hex`. When performing monetary I/O, `std::put_money()` prefixes values with the locale-dependent currency symbol, and `std::get_money()` requires a currency symbol prefix.
`showpoint`	Always use a locale-dependent decimal separator character for floating-point I/O, even if the decimal part is zero.
`showpos`	Use a + character for non-negative numeric I/O.
`skipws`	Instructs all formatted input operations (explained later) to skip leading whitespace.
`unitbuf`	Forces output to be flushed after each output operation. We refer to the section on `std::ostream` for a discussion on flushing.
`uppercase`	Instructs floating-point and hexadecimal integer output operations to use uppercase letters instead of lowercase ones.

Table 5-2. *`std::ios_base::fmtflags` Masks Defined in `<ios>`*

Flag	Description
`basefield`	`dec | oct | hex`
`adjustfield`	`left | right | internal`
`floatfield`	`scientific | fixed`

The initial formatting settings for streams are as follows (technically these are set by `std::ios`, the stream base class directly deriving from `std::ios_base`):

- Formatting flags are set to `skipws | dec`.
- Precision is set to 6.
- The field width is set to 0.
- The fill character is set to the space character (`' '`).

■ **Note** Next to the text formatting functions discussed in this section, the `ios_base` class also offers a set of more advanced member functions to facilitate the creation of user-defined I/O manipulators. We discuss these later in this chapter.

I/O Manipulators **<ios>, <iomanip>**

Manipulators allow you to change flags using `operator<<` and `operator>>` instead of `flags(fmtflags)` or `setf()`.

The `<ios>` header defines I/O manipulators in the global `std` scope for all the flags defined in Table 5-1: `std::scientific`, `std::left`, and so on. For flags that are part of a mask defined in Table 5-2, the I/O manipulator uses that mask. For example, `std::dec` actually calls `ios_base::setf(dec, basefield)`.

For `boolalpha`, `showbase`, `showpoint`, `showpos`, `skipws`, `uppercase`, and `unitbuf`, negative manipulators are available as well, which have the same name but are prefixed with `no`: for example, `std::noboolalpha`.

In addition to `std::fixed` and `scientific`, there are also `std::hexfloat` (`scientific | fixed`) and `std::defaultfloat` (no `floatfield` flags set) manipulators.

Additionally, the `<iomanip>` header defines the following manipulators:

Manipulator	Description
`setiosflags(fmtflags)` `resetiosflags(fmtflags)`	Sets/unsets the given `fmtflags`.
`setbase(int)`	Changes the base used for integer I/O. A value other than `16` (`hex`), `8` (`oct`), or `10` (`dec`) sets the base to `10`.
`setfill(char)`	Changes the fill character. See the example later.
`setprecision(int)`	Changes the number of decimal places for floating-point output as if set with `ios_base::precision()`.
`setw(int)`	Sets the width of the next field. See the example.

(*continued*)

Manipulator	Description
`get_money(m&,bool=false)` `put_money(m&,bool=false)`	Reads or writes a monetary value. If the Boolean is `true`, use international currency strings (e.g., `"USD "`); otherwise use currency symbols (e.g., `"$"`). The type of `m` can be either `std::string` or `long double`. See Chapter 6 for more details on monetary formatting.
`get_time(tm*, char*)` `put_time(tm*, char*)`	Reads or writes a date/time. The formatting is the same as for `std::strftime()`, discussed in Chapter 2.
`quoted(s, char='"',` `char='\\')`	Reads or writes a quoted string `s`. During output, quote characters (by default `'"'`s) are added around `s`, and every occurrence of the quote and escape characters in `s` is escaped (the default escape character is `'\\'`). When inputting the inverse transformation is applied. An example of this manipulator is given in the section on how to implement your own `operator<<` and `operator>>` later in this chapter.

All I/O manipulators except `std::setw()` are again *sticky*: once streamed, they impact all subsequent I/O operations on the same stream. `setw()` does not stick, though: this manipulator needs to be repeated once per value you want to stream with a nondefault width formatting. The next section contains an example of this.

Example

This code snippet additionally needs `<locale>`:

```
using namespace std;
cout.imbue(locale(""));  // Use the user's preferred locale, see Chapter 6
cout << setfill('_') << hex << showbase;
cout << "Left:     " << left << setw(7) << put_money(123) << '\n';
cout << "Right:    " << right << setw(7) << put_money(123) << '\n';
cout << "Internal: " << internal;
cout.width(7);
cout << 123 << '\n';
```

On an American system, the output is as follows (`hex` does not impact `put_money()`):

```
Left:     $1.23__
Right:    __$1.23
Internal: 0x___7b
```

Had we not repeated the nonsticky `setw(7)` manipulator before outputting the right-aligned value, the output of that line would have been `"Right: $1.23"` (without the extra padding of two underscores).

std::ios <ios>

The ios class defined in <ios> inherits from ios_base and provides, among other things, a number of methods to inspect and modify the *state* of a stream, which is a bitwise combination of the state flags listed in Table 5-3.

Table 5-3. *std::ios_base::iostate State Constants Defined in <ios>*

iostate	Description
goodbit	The stream is not in any error state. No bits are set: i.e., the state is 0.
badbit	The stream is in an unrecoverable error state.
failbit	An input or output operation failed. For example, reading a numerical value into an integer could cause the failbit to be set if the numerical value overflows the integer.
eofbit	The stream is at its end. This bit is only set by input operations.

Stream State

The following state-related methods are provided:

Method	Description
good() eof() bad() fail()	Returns true if, respectively, neither badbit, nor failbit, nor eofbit is set, the eofbit is set, the badbit is set, or either the failbit or badbit is set
operator!()	Equivalent to fail()
operator bool()	Equivalent to !fail() (this casting operator is marked explicit)
rdstate()	Returns the current ios_base::iostate state
clear(state)	Changes the state of the stream to the given one if a valid stream buffer is attached (see later); otherwise sets it to state | badbit
setstate(state)	Calls clear(state | rdstate())

■ **Caution** Do not use good() to determine whether one or more stream input operations succeeded (and the data you tried to read is thus safe to use):

```
// Read data from the input_stream stream...
if (input_stream.good())
   // Use the data that was read from input_stream
```

While this looks reasonable, `good()` fails if the stream was at its end—that is, if `eofbit` is set. Most of the time, you should therefore write the following instead:

```
if (input_stream)       //or: if (!input_stream.fail())
   // Use the data that was read from input_stream
```

You should use `good()` to determine whether a stream can be used for further I/O, like so (we therefore refer to `good()` as the *good-to-go* check):

```
if (my_stream.good())
   // Perform (additional) I/O operations with my_stream
```

The default initialization of `std::ios` sets the state to `goodbit` if there is a valid stream buffer attached (see later), or `badbit` otherwise.

Error Handling

By default, stream operations report errors by setting the state bits (`good`, `bad`, `fail`, and `eof`) of the stream, but they do not throw exceptions. Exceptions can be enabled, though, with the `exceptions()` method. It either returns the current exception mask or accepts one. This mask is a bitwise combination of `std::ios_base::iostate` state flags (see Table 5-3). For each state flag in the mask that is set to 1, the stream throws an exception when that state bit is set for the stream.

For example, the following code tries to open a nonexistent file using a file stream (defined in `<fstream>`, as explained in detail later in this chapter). No exceptions are thrown; only the fail bit of the stream is set to 1:

```
std::ifstream in("nonexistent_file.ext");
std::cout << in.fail() << std::endl;     // 1
```

If you want to use exceptions instead, the code can be rewritten as follows:

```
std::ifstream in("nonexistent_file.ext");
try {
  in.exceptions(std::ios_base::failbit); // Raise exceptions on failure
} catch (const std::ios_base::failure& exception) {
  std::cout << exception.what() << std::endl;
}
```

A possible output is

```
ios_base::failbit set: iostream stream error
```

Other Methods

Besides state-related methods, `ios` also defines the following methods

Method	Description
`fill()` `fill(char)`	Returns the current fill character, or changes it while returning the old one. To change it, you can also use the `setfill()` manipulator.
`copyfmt()`	Copies all formatting information, the `locale`, callbacks, and exception mask from another `ios` instance. The state is not copied.
`tie()` `tie(ostream*)`	Returns the currently tied output stream as a `std::ostream*` pointer (`nullptr` if there's no tied stream), or replaces it while returning the old one (if any). The output stream tied to the `this` stream is flushed each time an input or output operation is performed on the `this` stream. Flushing is discussed in the section on `std::ostream`.
`narrow()` `widen()`	Converts a wide character to its narrow equivalent or vice versa, in a locale-specific manner. See Chapter 6 for details on locales.

std::ostream `<ostream>`

The `ostream` class supports formatted and unformatted output to `char`-based streams. Formatted output means the format of what is written can be influenced by formatting options, such as the width of a field, the number of decimal digits for floating-point numbers, and so on. Formatted output is generally also influenced by the stream's `locale`, as explained in Chapter 6. Unformatted output entails simply writing characters or character buffers as is.

`ostream` provides a `swap()` method and the following high-level output operations. If no return type is mentioned, the operation returns an `ostream&`, allowing operations to be chained:

Operation	Description
`operator<<`	Writes formatted data to the stream.
`put(char)` `write(const char*, n)`	Writes a single character or `n` characters unformatted to the stream.

`fpos tellp()`
`seekp(pos)`
`seekp(off, dir)`
	Returns or changes the current position in the stream. The `p` in these method names is shorthand for `put` and denotes that these methods are working on an output stream. `seekp()` accepts either an absolute position (`fpos`) or an offset (`streamoff`) and a direction (`seekdir`: see Table 5-4) in which to start the offset.

`flush()`	Forcefully flushes the buffer to the target. Flushing and buffering are discussed in further detail after Table 5-4.

Table 5-4. *`std::ios_base::seekdir` Constants Defined in `<ios>`*

seekdir	Description
`beg`	The beginning of the stream
`end`	The end of the stream
`cur`	The current position in the stream

For performance reasons, output streams do not always directly write to their target outputs, but instead first write to some in-memory buffer (see also later) until some larger block of data is ready to be written out all at once. Especially for output to a hard disk drive, for instance, writing fewer, adequately sized bigger blocks of data is far more efficient than writing many small blocks in a row.

I/O Manipulators

`<ostream>` also defines the following extra I/O manipulators:

Manipulator	Description
`ends`	Writes `'\0'` (null character) to the stream
`flush`	Flushes the stream. Same as calling `flush()` on the `ostream`
`endl`	Writes `widen('\n')` to the stream and flushes it

■ **Caution** Be wary of `endl`, especially when outputting, for instance, data in a loop. Because each use of `endl` flushes the stream, using it repeatedly and prematurely may hamper buffering and therefore hurt performance. It is often better to just stream `'\n'` characters and to only explicitly flush (if ever) after a larger amount of output.

Global Output Streams `<iostream>`

The `<iostream>` header provides the following global `ostream` instances:

- `cout` / `wcout`: Outputs to the standard C output stream, `stdout`
- `cerr` / `wcerr`: Outputs to the standard C error stream, `stderr`
- `clog` / `wclog`: Outputs to the standard C error stream, `stderr`

`(w)cerr` and `(w)clog` are intended for output of errors and logging information, respectively. Their destinations are implementation specific.

`(w)cout` is automatically tied to `(w)cin`. This means an input operation on `(w)cin` causes `(w)cout` to flush its buffers. `(w)cout` is also automatically tied to `(w)cerr`, so any output operation on `(w)cerr` causes `(w)cout` to flush.

`std::ios_base` provides a static method called `sync_with_stdio()` to synchronize these global `ostream`s with the underlying C streams after each output operation. This ensures that they both use the same buffers, allowing you to safely mix C++ and C-style output. It also guarantees that the standard streams are thread-safe: that is, there are no data races. Character interleaving remains possible, though.

■ **Note** When working with the standard streams `cout`, `cerr`, `clog`, and `cin` (discussed later), you do not have to take care of platform-dependent end-of-line characters. For example, on Windows, a line usually ends with `\r\n`, whereas on Linux it ends with `\n`. However, the translation happens automatically for you, so you can just always use `\n`.

Example

The following example demonstrates the three different methods of output:

```
std::cout << "PI = " << 3.1415 << std::endl;
std::cout.put('\t');
std::cout.write("C++", 3);
```

std::istream **<istream>**

The `istream` class supports formatted and unformatted input from `char`-based streams. It provides `swap()` and the following high-level input operations. Unless otherwise specified, the operation returns an `istream&`, which facilitates chaining:

Operation	Description
`operator>>`	Reads formatted data from the stream. All other input operations work with unformatted data.

`get(char*, count [, delim])`
`getline(char*, count [, delim])`
`read(char*, count)`
	Reads `count` characters from the stream and stores them in a `char*` buffer. A terminating null character (`'\0'`) is automatically added by `get()` and `getline()`, but not by `read()`. For the first two, input stops when encountering the delimiter, by default `'\n'`. `get()` does not extract the delimiter from the stream, but `getline()` does. The delimiter is never stored in the `char*` buffer.

`streamsize readsome( char*, count)`	Reads at most `count` characters that are immediately available into a given `char*` buffer. These are the characters the underlying stream buffer (discussed later) can return without having to wait for them, used, for instance, to read data from asynchronous sources without blocking. Returns the number of extracted characters.

(*continued*)

Operation	Description

`get(char&)`
`int get()`
`int peek()`
	Reads a single character from the stream. The first version stores the read character in a `char` reference. The last two return an integer that is either a valid read character or `EOF` if no characters are available. `peek()` does not remove the character from the stream.

`unget()` `putback(char)`	Puts the last read character or a given one on the stream so it is available for the next read operation.
`ignore([count [,delim]])`	Reads `count` characters (`1` by default) from the stream or until a given delimiting character is encountered (`eof` by default) and discards them. The delimiter is removed as well.
`streamsize gcount()`	Returns the number of characters that were extracted by the last unformatted input operation: `get()`, `getline()`, `read()`, `readsome()`, `peek()`, `unget()`, `putback()`, or `ignore()`.

`fpos tellg()`
`seekg(pos)`
`seekg(off, dir)`
	Returns or changes the current position in the stream. The `g` is shorthand for `get` and denotes that these methods are working on an input stream. `seekg()` accepts either an absolute position (`fpos`) or an offset (`streamoff`) and a direction (`seekdir`: see Table 5-4) in which to start the offset.

`int sync()`	Synchronizes the input stream with the underlying stream buffer (discussed later). This is an advanced, rarely used method.

I/O Manipulators

`<istream>` also defines the following extra I/O manipulator:

Manipulator	Description
`ws`	Discards any whitespace currently in the stream

Global Input Streams `<iostream>`

The `<iostream>` header provides the following global `istream` instances:

- `cin` / `wcin`: Reads from the standard C input stream, `stdin`

The `ios_base::sync_with_stdio()` function affects `(w)cin` as well. See the explanation given for `cout`, `cerr`, and `clog` earlier.

Example

As explained earlier, `istream` provides a `getline()` method to extract characters. Unfortunately, you have to pass it a `char*` buffer of proper size. The `<string>` header defines a `std::getline()` method that is easier to use and that accepts a `std::string` as target buffer. The following example illustrates its use:

```
int anInt;
double aDouble;
std::cout << "Enter an integer followed by some whitespace\n"
          << "and a double, and press enter: ";
std::cin >> anInt >> aDouble;
std::cout << "You entered: ";
std::cout << "Integer = " << anInt << ", Double = " << aDouble
          << std::endl;

std::string message;
std::cout << "Enter a string. End input with a * and enter: ";
std::getline(std::cin >> std::ws, message, '*');
std::cout << "You entered: '" << message << "'" << std::endl;
```

Here is a possible output of this program:

```
Enter an integer followed by some whitespace
and a double, and press enter: 1    3.2↵
You entered: Integer = 1, Double = 3.2
Enter a string. End input with a * and enter: This is↵
a multiline test*↵
You entered: 'This is
a multiline test'
```

std::iostream **<istream>**

The `iostream` class, defined in `<istream>` (not in `<iostream>`!), inherits from both `ostream` and `istream` and provides high-level input and output operations. It keeps track of two independent positions in the stream: an input and an output position. This is the reason `ostream` has `tellp()` and `seekp()` methods, whereas `istream` has `tellg()` and `seekg()`: `iostream` contains all four, so they need a different name. It does not provide additional functionality beyond what is inherited.

String Streams **<sstream>**

String streams allow you to use stream I/O on `std::string`s (`std::wstring`s for the wide character versions of the streams). The library provides `istringstream` (input, inherits

from istream), ostringstream (output, inherits from ostream), and stringstream (input and output, inherits from iostream). See Figure 5-1 for the inheritance chart. All three classes have a similar set of constructors:

- [i|o]stringstream(ios_base::openmode): Constructs a new string stream with the given openmode, a bitwise combination of the flags defined in Table 5-5
- [i|o]stringstream(string&, ios_base::openmode): Constructs a new string stream with a copy of the given string as initial stream contents and with the given openmode
- [i|o]stringstream([i|o]stringstream&&): Move constructor

Table 5-5. *std::ios_base::openmode Constants Defined in <ios>*

openmode	Description
app	Short for append. Seeks to the end of the stream before each write.
binary	A stream opened in binary mode. If not specified, the stream is opened in text mode. See the "File Streams" section for the difference.
in / out	A stream opened for reading/writing, respectively.
trunc	Removes the contents of the stream after opening it.
ate	Seeks to the end of the stream after opening it.

The openmode in the first two constructors has a default: out for ostringstream, in for istringstream, and out|in for stringstream. For ostringstream and istringstream, the given openmode is always combined with the default one; for example, for ostringstream, the actual openmode is *given_openmode*|ios_base::out.

All three classes add only two methods:

- string str(): Returns a copy of the underlying string buffer
- void str(string&): Copies the given string to the underlying string buffer, replacing any previous content of the buffer

Example

```
std::ostringstream oss;
oss << 123 << " " << 3.1415;
std::string myString = oss.str();
std::cout << "ostringstream contains: '" << myString << "'" << std::endl;

std::istringstream iss(myString);
int myInt; double myDouble;
iss >> myInt >> myDouble;
std::cout << "int = " << myInt << ", double = " << myDouble << std::endl;
```

File Streams

<fstream>

File streams allow you to use stream I/O on files. The library provides an `ifstream` (input, inherits from `istream`), `ofstream` (output, inherits from `ostream`), and `fstream` (input and output, inherits from `iostream`). See Figure 5-1 for the inheritance chart. All three classes have a similar set of constructors:

- `[i|o]fstream(filename, ios_base::openmode)`: Constructs a file stream and opens the given file with the given `openmode`. The file can be specified as a `const char*` or a `std::string&`. Sets the state to `failbit` if opening the file fails.
- `[i|o]fstream([i|o]fstream&&)`: Move constructor.

■ **Tip** File streams can also be opened with a `path` of the file system library discussed later in this chapter, because a `path` converts implicitly to a `string`.

All three classes add the following methods:

- `open(filename, ios_base::openmode)`: Opens a file similar to the first constructor.
- `is_open()`: Returns `true` if a file is opened for input and/or output.
- `close()`: Closes the currently opened file. Any pending output is written out first.

The `openmode` (see Table 5-5) in the constructors and in the `open()` method has a default: `out` for `ofstream`, `in` for `ifstream`, and `out | in` for `fstream`. For `ofstream` and `ifstream`, the given `openmode` is always combined with the default one; for example, for `ofstream`, the actual `openmode` is *given_openmode* | `ios_base::out`.

If the `ios_base::in` flag is specified, whether or not in combination with `ios_base::out`, the file you are trying to open must already exist. The following code opens a file for input and output and creates the file if it does not exist yet:

```
std::string filename = "data.txt";
std::fstream fs(filename); // Default openmode=ios_base::in|ios_base::out
if (!fs.good()) {          // Fail bit will be set if file does not exist
   fs.clear();             // First clear the error state
   fs.open(filename, std::ios_base::out); // Create the file
   fs.close();          // Close and reopen the file for input and output
   fs.open(filename, std::ios_base::in | std::ios_base::out);
}
```

■ **Tip** To verify whether a given file exists, you may also use the file system library discussed later in this chapter.

If a file is opened in text mode, as opposed to binary mode, the library is allowed to translate certain special characters to match how the platform uses those. For example, on Windows, lines usually end with `\r\n`, whereas on Linux they usually end with `\n`. When a file is opened in text mode, you do not read/write the `\r` on Windows yourself; the library handles this translation for you.

The `fstream` class, supporting both input and output, handles the current position differently compared to other combined input and output streams, such as `stringstream`. A file stream has only one position, so the output and input positions are always the same.

■ **Tip** The destructor of a file stream automatically closes the file.

Example

The following example is similar to the example given earlier for string streams but uses a file instead. In this example, the `ofstream` is explicitly closed using `close()`, and the `ifstream` is implicitly closed by the destructor of `ifs`:

```
const std::string filename = "output.txt";
std::ofstream ofs(filename);
ofs << 123 << " " << 3.1415;
ofs.close();

std::ifstream ifs(filename);
int myInt; double myDouble;
ifs >> myInt >> myDouble;
std::cout << "int = " << myInt << ", double = " << myDouble << std::endl;
```

Streaming Custom Types

Custom << and >> Operators

You can write your own versions of the stream output and extraction operators `operator<<` and `operator>>`. What follows is an example of both operators for the `Person` class. It uses the `std::quoted()` manipulator to handle spaces in names. For the sake of the example, assume a valid `Person` object needs a nonempty first name:

```
std::ostream& operator<<(std::ostream& out, const Person& person) {
   return out << std::quoted(person.GetFirstName()) << ' '
              << std::quoted(person.GetLastName());
}

std::istream& operator>>(std::istream& in, Person& person) {
   std::string firstName, lastName;
   in >> std::quoted(firstName) >> std::quoted(lastName);
   if (firstName.empty())             // fail if invalid data is read
     in.setstate(std::ios::failbit);  // add fail bit
   else if (in)                       // only if reading succeeded
     person = Person(std::move(firstName), std::move(lastName));
   return in;
}
```

These operators can be used as follows (<sstream> is also required):

```
Person kurt("Kurt", "von Strohm");
std::stringstream ss;
ss << kurt;
std::cout << ss.str() << '\n';   // "Kurt" "von Strohm"
ss.seekg(0);                     // Seek back to beginning of stream
Person readBack;
ss >> readBack;
std::cout << readBack << '\n';   // "Kurt" "von Strohm"
```

Custom I/O Manipulators <ios>

All stream classes offer a set of functions that essentially allow you to add state to their instances (functionally equivalent to adding member variables). A typical use (of which we will give an example later) is to store values inside a stream from within a custom I/O manipulator, and then use these values in custom << and >> operators. Custom `locale` facets (see Chapter 6) could use this state as well.

Here is a brief overview of these functions (all defined by `ios_base`):

Method	Description
`int xalloc()`	Static member function that generates a program-wide unique index to pass to `iword()` and/or `pword()`. This function is thread-safe.
`long& iword(int)` `void*& pword(int)`	Member functions that return a reference to a `long` variable or `void*` pointer owned by the stream instance. The integer index you pass must be obtained first from `xalloc()`.
`register_callback(` `event_callback,int)`	Adds a callback function (passed as a function pointer) that will be called when either `imbue()`, `copyfmt()`, or `~ios_base()` gets invoked. Callbacks cannot be unregistered. The integer value passed as a second argument (typically a `xalloc()` index) is passed along with any invocation of the callback.

The first call to `iword()` / `pword()` with a given index results in a reference to a zero-initialized value (`0L` / `nullptr`). Subsequent calls to `iword()` / `pword()` may invalidate references returned earlier by the same function (if reallocation occurs). The values associated with a given index, though, always remain retained (only `copyfmt()` replaces them).

The `iword()` and `pword()` members are backed by disjoint arrays. This means the same `xalloc()` index can thus be reused for both, and you will get references to different memory addresses.

The functions registered with `register_callback()` must return `void` and accept the same parameter types as shown here:

```
void my_callback(std::ios_base::event, std::ios_base&, int);
```

When such a callback function is invoked by the stream, the third argument is the integer value that you passed to `register_callback()` when registering the callback (as noted earlier, this value will often be an index obtained by `xalloc()`). The `std::ios_base::event` argument will be one of the enumeration values listed in the following table. Each enumeration value corresponds to the moments at which registered callbacks are invoked by a stream:

std::ios_base::event	For Callbacks Invoked…
`erase_event`	…from `~ios_base()` or `basic_ios::copyfmt()` (the latter before the `iword()` and `pword()` values are overwritten)
`imbue_event`	…after a new locale has been installed by `imbue()`
`copyfmt_event`	…after `basic_ios::copyfmt()` has replaced all `iword()` and `pword()` values (but before the state bits are replaced)

■ **Tip** You can use `pword()` to store a pointer to dynamically allocated memory, which you then release from inside a callback registered with `register_callback()`.

Example

In this example, we create `dot()` and `nodot()` I/O manipulators to toggle between printing the full first name of a `Person` (the default) and only printing the first initial. Here is the code:

```
const int dot_xalloc = std::ios_base::xalloc(); // global constant

// I/O manipulators that toggle the 'dot' field of a stream
std::ios_base& dot(std::ios_base& stream)
  { stream.iword(dot_xalloc) = 1; return stream; }
std::ios_base& nodot(std::ios_base& stream)
  { stream.iword(dot_xalloc) = 0; return stream; }

// Output stream operator that 'dots' the first name if requested
std::ostream& operator<<(std::ostream& out, const Person& person) {
   if (out.iword(dot_xalloc))
      out << person.GetFirstName().front() << '.';
   else
      out << person.GetFirstName();
   return out << ' ' << person.GetLastName();
}
```

The `dot` manipulator can then be used as follows:

```
Person person("Hubert", "Gruber");
std::cout << person << '\n' << dot << person << std::endl;
```

which will output

```
Hubert Gruber
H. Gruber
```

■ **Tip** A portable output stream manipulator that accepts arguments looks as follows (input stream manipulators are analogous):

```
struct my_manip {
   my_manip(/* Parameters... */) : /* Store in members... */ {}
   /* Member variables to store arguments ...*/
};
std::ostream& operator<<(std::ostream& out, const my_manip& m) {
   /* ... use members stored in m to manipulate out */
   return out;
}
```

You can then use `my_manip` just like you would the manipulators of `<iomanip>`:

```
std::cout << my_manip(/* Arguments... */) << /* ... */;
```

Stream Iterators <iterator>

The <iterator> header defines two stream iterators, istream_iterator and ostream_iterator, in addition to the other iterators discussed in Chapters 3 and 4.

std::ostream_iterator

The ostream_iterator is an output iterator capable of outputting a sequence of objects of a certain type to an ostream using operator<<. The type of the objects to output is specified as a template type parameter. There is one constructor that accepts a reference to the ostream to use and an optional delimiter that is written to the stream after each output.

Stream iterators are very powerful in combination with the algorithms discussed in Chapter 4. As an example, the following code snippet writes a vector of doubles to the console using the std::copy() algorithm, where each double is followed by a tab character (additionally requires <vector> and <algorithm>):

```
std::vector vec{ 1.11, 2.22, 3.33, 4.44 };
std::copy(cbegin(vec), cend(vec),
          std::ostream_iterator<double>(std::cout, "\t"));
```

std::istream_iterator

The istream_iterator is an input iterator capable of iterating over objects of a certain type in an istream by extracting them one by one using operator>>. The type of the objects to extract from the stream is specified as a template type parameter. There are three constructors:

- istream_iterator(): The default constructor, which results in an iterator pointing to the end of the stream
- istream_iterator(istream&): Constructs an iterator that extracts objects from the given istream
- istream_iterator(istream_iterator&): Copy constructor

Just like an ostream_iterator, istream_iterators are very powerful in combination with algorithms. The following example uses the for_each() algorithm in combination with an istream_iterator to read an unspecified number of double values from the standard input stream and sum them to calculate the average (additionally needs <algorithm>):

```
std::istream_iterator<double> begin(std::cin), end;
double sum = 0.0; int count = 0;
std::for_each(begin, end, [&](double value){ sum += value; ++count;});
std::cout << sum / count << std::endl;
```

Input is terminated by pressing Ctrl+Z on Windows or Ctrl+D on Linux, followed by Enter.

Our second example uses both an `istream_iterator` to read an unspecified number of `double`s from the console and an `ostream_iterator` to write the read `double`s to a `stringstream` separated by tabs (additionally needs `<sstream>` and `<algorithm>`):

```
std::ostringstream oss;
std::istream_iterator<double> begin(std::cin), end;
std::copy(begin, end, std::ostream_iterator<double>(oss, "\t"));
std::cout << oss.str() << std::endl;
```

Stream Buffers `<streambuf>`

As noted earlier, stream classes do not work directly with a target such as a string in memory, a file on disk, and so on. Instead, they use the concept of *stream buffers*, defined by `std::basic_streambuf<CharType>`. Two type aliases are provided, `std::streambuf` and `std::wstreambuf`, where the template type argument is, respectively, `char` or `wchar_t`. File streams use `std::(w)filebuf` and string streams `std::(w)stringbuf`, both inheriting from `(w)streambuf`.

Each stream has a stream buffer associated with it to which you can get a pointer with `rdbuf()`. A call to `rdbuf(streambuf*)` returns the current associated stream buffer and changes it to the given one.

Stream buffers can be used to write a stream-redirector class that redirects one stream to another stream. As a basic example, the following code snippet redirects all `std::cout` output to a file (additionally needs `<fstream>`):

```
std::ofstream file("output.txt");
auto oldCoutBuf = std::cout.rdbuf(file.rdbuf()); // Redirect cout to file
std::cout << "Some output" << '\n';  // Write to file
std::cout.rdbuf(oldCoutBuf);         // Restore the old cout buffer!
```

■ **Caution** When changing the buffer for one of the standard streams, do not forget to restore the old buffer before terminating the application, as is done in the previous example. Otherwise your code may crash with certain library implementations.

It can also be used to implement a tee class that redirects output to two or more target streams. Another use is to easily read an entire file:

```
std::ifstream ifs("test.txt");
std::stringstream buffer;
buffer << ifs.rdbuf();
```

The exact behavior of stream buffers is implementation dependent. Working directly with stream buffers is an advanced topic that we cannot discuss further in detail due to page constraints.

File Systems <filesystem>

The <filesystem> library allows you to determine which files are present on your file system, to inspect and manipulate their properties, and to create, copy, and delete such files. Before we delve into the functionality of this library, though, we first establish some essential terminology related to files, paths, and pathnames.

■ **Note** The <filesystem> library does not provide facilities to read from or write to files: for that you use the file streams discussed earlier in this chapter.

Files, Paths, and Pathnames

A *file* is an object within a file system that holds data. In the terminology of the library, a *directory* is a *file* as well. What most would refer to as a 'file'—as in a text file, an executable, or a multimedia file—is called a *regular file*. As discussed later, implementations typically support other file types as well, such as links and pipes.

A *path* is a sequence of elements that identifies the location of a file within the file system. It could identify an existing file or a nonexistent file that you are about to create. A *pathname* is a textual representation of the path.

Pathnames can be expressed either in the *native format* of the underlying file system or the portable, so-called *generic format* of the library. This generic format corresponds to that of the POSIX standard (in fact: on POSIX-based systems, there is generally no difference between the native and generic pathname format). When expressed in the generic format, a pathname consists of the following components:

```
[root name][root directory][relative path]
```

where

- **[root name]** is an optional root designation for file systems with multiple roots.
- **[root directory]** is an optional *directory separator*.
- **[relative path]** is a sequence of zero or more *filenames*, separated by *directory separators*.

On Windows, `"D:"` or `"\\server1"` could be examples of root names. POSIX-like file systems generally only have one root. For such systems, pathnames have no root name. A directory separator is either `"/"` or the preferred separator of the operating system (for instance, `"\"` on Windows). This preferred separator character can be obtained from `std::filesystem::path::preferred_separator`.

■ **Note** The native Windows file system API accepts both forward slash `"/"` and backslash `"\"` as directory separators in pathnames, as should all Windows components and programs. The Visual Studio `<filesystem>` implementation accepts both as well.

Paths can be *absolute* or *relative*. An absolute path unambiguously identifies the location of a file, whereas a relative one does so only starting from an additional starting location. Which elements exactly determine that a pathname is absolute or not is implementation dependent.

An example pathname on Windows is `"C:\Windows\System32\notepad.exe"`. It represents an absolute path. Omitting the root name `"C:"` would make it a relative path (still with root directory `"\"`). An example pathname for a Linux file system is `"/var/log/kern.log"`. Even though it has no root name, it is still an absolute path.

Two special (relative) pathnames are `"."`, representing the current directory, and `".."`, representing the parent directory.

Error Reporting

Functions of the file system library that interact with the underlying file system can typically fail—and if they do, they report failures in one of two ways:

- Overloads without a `std::error_code&` output parameter throw a `std::filesystem::filesystem_error` exception when an error occur.
- Overloads with a `std::error_code&` output parameter store the error in the given error instead.

Functions can also throw a `bad_alloc` exception if a memory allocation error occurs (even the overloads with an `error_code&` output parameter), unless the function is marked as `noexcept`.

■ **Note** To save space, we will not show optional `std::error_code&` parameters in the remainder of this text. Know, though, that whenever we mention that a function may fail, both error reporting mechanisms will be present. Not all functions can fail though: none of the member functions of the `path` class we discuss next, for instance, can fail (they operate entirely at the level of pathname strings). Many non-member functions that work with `path`s can fail though, as do many directory listing operations.

The path Class

A `std::filesystem::path` represents a path and stores a pathname encoded using the *native pathname format* in the *native character encoding* of the underlying file system. `std::filesystem::path::value_type` is an alias for the character type used by this encoding and `std::filesystem::string_type` for `basic_string<value_type>`. All members that accept or return a pathname string are capable of converting from or to the generic pathname format and/or other character encodings when needed.

■ **Note** A `path`'s native pathname does not necessarily use (only) the preferred directory separator—on Windows, for instance, a valid native pathname may use a mixture of `'/'` and `'/'` separators. On Windows, `value_type` is normally `wchar_t`; on POSIX-based systems, it is generally `char`.

Construction and Assignment

You can create a `path` object from a given pathname using any of the constructors of the form `path(pathname[, locale][, format])`, where

- `pathname` is either a C-style string, a `std::basic_string`, a `basic_string_view`, or an iterator range into a character array. Various encodings are supported, as explained shortly.
- `locale`, if provided, is used to convert `pathname` to the native encoding of the file system.
- `format` is a value of the `std::filesystem::path::format` enumeration type. Possible values are `native_format`, `generic_format`, and `auto_format`. With `auto_format` (the default), the interpretation of `pathname` is implementation-defined. Typically, though, the constructor will inspect the contents of the given `pathname` to determine whether it is specified in either the native or generic pathname format.

If you do not provide a `locale`, the character type of `pathname` can be either `char`, `wchar_t`, `char16_t`, or `char32_t`. The pathname is then assumed to be encoded using the native narrow, native wide, UTF-16, or UTF-32 character encoding, respectively. If you do provide a `locale`, however, the character type of `pathname` must be `char`, and the native narrow encoding of the platform must be used.

UTF-8 pathnames are supported as well through `std::filesystem::u8path()`.[1] This non-member factory function creates a `path` from a given UTF-8 encoded string either of type `const char*`, `string`, or `string_view`, or passed as an iterator range. `u8path()` always uses `auto_format` and does not accept a `locale`.

---

[1] The `u8path()` factory function will be deprecated by C++20 with the introduction of the `char8_t` character type. With this addition, the normal `path` constructors will be updated to support UTF-8 encoded strings with type `char8_t`.

A path can be copied, moved, and swapped. A default-constructed path is *empty*. The empty() method checks whether a path contains a pathname or not.

You replace a path's pathname using either operator=() or assign(). These two methods accept the same string inputs and encodings as the constructors, but no locale or format parameter (auto_format is used).

Conversion to Strings

You can convert a path into a pathname string using any of the following functions:

Method	Description
c_str() native() operator string_type()	Direct access to the native pathname in its native encoding. c_str() returns a const value_type* pointer, native() a const string_type& reference.
string() wstring() u8string() u16string() u32string()	Returns a copy of the native pathname, encoded using respectively the native narrow encoding, the native wide encoding, UTF-8, UTF-16, and UTF-32. The character types used for the resulting basic_string<CharT> are respectively char, wchar_t, char,[2] char16_t, and char32_t.
generic_string() generic_wstring() generic_u8string() generic_u16string() generic_u32string()	Returns the pathname expressed in the generic format, encoded with the same encoding and character types as the previous set of functions. All native directory separators are replaced with '/'.
string<CharT>() generic_string<CharT>()	Member function templates with the same template type parameter list as basic_string, of which only the character type parameter CharT is non-optional. Return the same values as the corresponding regular functions for the same character type (where for char the native narrow encoding is used). These function templates also accept an optional std::allocator& argument (see Chapter 3 for allocators).

Decomposition

A path offers begin() and end() methods to iterate over its elements. For instance:

```
std::filesystem::path my_path("c:/Windows/notepad.exe");
for (const std::filesystem::path& element : my_path)
   std::cout << element << ", ";
```

[2] Will normally become char8_t in C++20.

On a Windows system, this example prints

```
"c:", "/", "Windows", "notepad.exe",
```

■ **Note** The stream insertion operator for `path`, `<<`, outputs the underlying native pathname surrounded with double quotes (as if by `std::quoted()`). The corresponding stream extraction operator, `>>`, also disregards any surrounding quotes, if present.

If a path ends with a non-root directory separator, an empty `path` element is appended to the iteration. Repeating the previous example with pathname `"c:/Windows/"` therefore gives the following output:

```
"c:", "/", "Windows", "",
```

Other members that decompose a `path` are listed in the following table. Also depicted are the pathnames of the `path` objects that these methods return when invoked on three example `path`s (the first two are for a Windows file system, the third for a POSIX system).

Method	`c:\Windows\notepad.exe`	`d:\`	`gonzo/.profile`
`root_name()`	`c:`	`d:`	*(empty)*
`root_directory()`	`\`	`\`	*(empty)*
`root_path()`	`c:\`	`d:\`	*(empty)*
`relative_path()`	`Windows\notepad.exe`	*(empty)*	`gonzo/.profile`
`parent_path()`	`c:\Windows`	`d:\`	`gonzo`
`filename()`	`notepad.exe`	*(empty)*	`.profile`
`extension()`	`.exe`	*(empty)*	*(empty)*
`stem()`	`notepad`	*(empty)*	`.profile`

As seen from these examples, a `path`'s extension starts with the rightmost period in the filename. If the first character of the filename is a period, however, that period is ignored. The special filenames `"."` and `".."` have no extension either. Also notice that the parent `path` of a `path` that consists solely of root elements is the `path` itself.

For each of the methods in the previous table, there is also a `has_x()` method that returns a Boolean (i.e., `has_root_name()`, `has_filename()`, etc.).

■ **Note** Because directories are files as well, `filename()` will return the name of a directory if given a `path` that refers to a directory. Do recall though that if a pathname ends with a directory separator, an extra empty element is added to the path. In that case, `filename()` will return an empty `path` (and `has_filename()` will return `false`). Here is an example:

```
using std::filesystem::path;
std::cout << path(R"(c:\Temp)").filename() << std::endl;  // "Temp"
std::cout << path(R"(c:\Temp\)").filename() << std::endl; // ""
```

Composition

There are two mechanisms for composing `paths`:

1. You can *append* `paths` through `operator/=`, `operator/`, or the `append()` method.
2. You can *concatenate* `paths` through `operator+=` or `concat()` (there is no `operator+` for `paths`).

The key difference is that `/=`, `/`, and `append()` insert a (preferred) directory separator between two pathnames if the first does not already end with a separator, whereas `+=` and `concat()` simply concatenate pathnames without inserting any separators. All composition methods accept `path` objects, as well as the same string inputs as the `path` constructors (but without `locale` or `format` arguments).

Here is an example that appends and concatenates pathnames:

```
auto get_full_path(int ind) {
   auto temp_dir = std::filesystem::path(LR"(c:\temp)");
   auto full_path = temp_dir/u8"file";       // or "file", L"file", etc.
   full_path += std::to_string(ind) + ".log";// or full_path.append(...);
   return full_path;
}
```

Assuming `ind` equals `10`, the preceding function produces a `path` `full_path` that contains `"c:\temp\file10.log"` as pathname.

Concatenation blindly concatenates pathnames, even if this results in an invalid path. For instance, with `temp_dir` defined as before, the contrived statement '`temp_dir += temp_dir;`' would result in the invalid pathname `"c:\tempc:\temp"`.

Appending two `paths` `p1` and `p2`, on the other hand, never results in an invalid `path` and has some special semantics when mixing absolute and relative paths:

- If `p2` is either an absolute path or a path with a nonempty root name different from that of `p1`, the result is `p2`.
- Otherwise, the native format pathname of `p2`, without its root name, is appended to that of `p1`. If `p2` has a root directory, any root directory and entire relative path of `p1` are removed first.

```
using std::filesystem::path;
std::cout << path("foo") / "bar" << '\n';      // "foo/bar" or "foo\\bar"
std::cout << path("foo") / "/bar" << '\n';     // "/bar"
std::cout << path("foo") / "c:/bar" << '\n';   // "c:/bar"      (Windows)
std::cout << path("c:/foo") / "d:bar" << '\n'; // "d:bar"       (Windows)
std::cout << path("c:/foo") / "c:bar" << '\n'; // "c:/foo\\bar" (Windows)
std::cout << path("c:/foo") / "/bar" << '\n';  // "c:/bar"      (Windows)
```

■ **Note** The double backslashes originate from the use of `std::quoted()` by the stream insertion operator for `path`s and the fact that backslash is both the preferred directory separator on Windows and the default escape character of `std::quoted()`.

Modification

The following table lists some methods that modify a given `path` object.

Method	Description
`clear()`	Turns the `path` into an empty `path`.
`make_preferred()`	Converts all generic directory separators, `'/'`, to the native preferred separator (`'\'` on Windows).
`remove_filename()`	Removes the filename part of the `path` (if any).
`replace_filename(p)`	Removes the filename part of the `path` (if any) and appends `p` as if by `/=`. The argument `p` can be any `path`, not just a single filename.
`replace_extension(p={})`	Removes the extension part of the `path` and concatenates `p` as if by `+=`. If `p` is empty (or omitted), this effectively removes the extension. An extra dot is inserted before `p` if `p` is nonempty and does not itself begin with a dot.

File Links

A file system *link* associates a filename with a file. Several links can refer to the same file. In general, there are two types of links: hard links and symbolic links. A *hard link* can be thought of as a `shared_ptr`: if the last hard link to a file is removed, the file itself is removed as well. It is also much like a C++ reference, as a hard link is generally indistinguishable from the real file. A *symbolic link*, on the other hand, acts more like a raw pointer: it is simply a path stored by the file system, referring to another file. If that file does not exist, the symbolic link is said to be *dangling*. The term symbolic link is commonly shortened to *symlink*.

Hard and symbolic links are supported by most platforms. All POSIX-compliant operating systems support them, as does the NTFS file system used by most Windows systems.

■ **Caution** *Shortcuts* in Windows are not file system links. Applications such as Windows Explorer only treat such files as if they are, but to the file system they are simply regular files with extension `.lnk`. While less known, a Windows user can create actual symbolic or hard links, though, either with the `mklink` Command Prompt command or with various third-party applications and Windows Explorer extensions.

The following table lists all functions related to filesystem links:

Function	Description

`create_hard_link(t,l)`
`create_symlink(t,l)`
`create_directory_symlink(t,l)`
	Creates a hard/symbolic link at `path l` that refers to the file at target `path t`. For symlinks, the target file does not have to exist. Portable code should use `create_directory_symlink()` to create symlinks to directories, even though on some platforms (and on POSIX systems in particular) you may be able to do so with `create_symlink()` as well.

`read_symlink(link)`	Returns the target `path` of a symbolic link. Fails if the given `path` does not refer to a symlink file.
`copy_symlink(from,to)`	Creates a new symlink at `path to` that refers to the same target as the symlink at `from`. Fails if the latter `path` does not correspond to a symlink file.
`hard_link_count(path)`	Returns the number of hard links to the file system object identified by `path`.
`is_symlink(path)`	Returns true if `path` is known to identify a symlink; false if it does not. See also Table 5-6.

Path Normalization

A generic pathname in *normal form* uses only the preferred directory separator and does not contain any redundant directory separators or `"."` and `".."` elements. For instance, on Windows, the normal form of `"/../Windows//./Temp/../win.ini"` is `"\Windows\win.ini"`. A path is in normal form if its generic pathname is in normal form.

The following functions generate paths in normal, or near-normal, form:

Function	Description
`p.lexically_normal()`	Returns a path whose generic pathname is the normal form of the generic pathname of p. This member does not access the file system, but operates purely *lexically*—i.e., by inspecting and manipulating pathname strings. This implies that symbolic links are not resolved during normalization: see `canonical()` for a variant that does do this.
`canonical(p)`	Returns an absolute path that refers to the same file as p, but whose generic pathname contains no symbolic links, `"."`, or `".."` elements. Note that the standard does not guarantee that the resulting path will be in normal form (its pathname could, for instance, contain nonpreferred or redundant directory separators). Fails if the path p does not exist.
`weakly_canonical(p)`	Converts the longest sequence of leading elements that form an existing path as if by `canonical()`, and then appends the remaining elements. The resulting path is in normal form.

■ **Note** Even though the normal form of an empty path is an empty path, normalization never produces an empty path when starting from a nonempty input. Instead, paths such as `"./"` and `"dir/.."` are normalized to `"."`.

The Current Working Directory

`std::filesystem::current_path()` returns the absolute path of the current *working directory* associated with the process. It typically starts out as the directory from which the application was launched, which may or may not correspond to the directory in which the executable is located. This is also the path that is used by default to resolve relative paths, as seen in the next subsection.

Absolute and Relative Paths

The following table assumes the existence of the following two paths for its examples:

```
std::filesystem::path abs("c:/Windows/notepad.exe");
std::filesystem::path rel("../log.txt");
```

Function	Returns…

`p.is_absolute()`
`p.is_relative()`
	Returns `true` if the path represents an absolute or relative path, respectively, and `false` otherwise. For example:
	`abs.is_absolute() == true`
	`rel.is_relative() == true`

`absolute(p)`	Returns `p` converted into an absolute path. The result depends on the current working directory. For instance, suppose the working directory is `"D:\temp\App1\x64"`, then `absolute(rel) == "D:\\temp\\App1\\log.txt"`

`p.lexically_relative(base)`	Returns `p` made relative to the given base `path`. For instance:
	`abs.lexically_relative("c:/")`
	`  == "Windows\\notepad.exe"`
	`abs.lexically_relative("c:/Windows/")`
	`  == "notepad.exe"`
	`abs.lexically_relative("c:/Windows/Temp")`
	`  == "..\\notepad.exe"`
	`abs.lexically_relative(abs) == "."`
	Unlike the non-member functions discussed later, this purely lexical operation does not follow symlinks. Returns an empty `path` if the root names do not match, if one of the paths is absolute while the other is not, or if `base` has a root directory while `p` does not.

`p.lexically_proximate(base)`	Returns `p.lexically_relative(base)` if that is not empty, otherwise returns `p`. For example:
	`abs.lexically_relative("d:/Temp/") == ""`
	`abs.lexically_proximate("d:/Temp/") == "c:/Windows/notepad.exe"`

`relative(p, base = current_path())`	Equivalent to
	`weakly_canonical(p)`
	`  .lexically_relative(weakly_canonical(base))`

`proximate(p, base = current_path())`	Equivalent to
	`weakly_canonical(p)`
	`  .lexically_proximate(weakly_canonical(base))`

■ **Caution** The `[lexically_]relative()` / `proximate()` functions all assume the given `base path` refers to a directory. The filename component of the pathname will thus always be assumed to be that of a directory, even if it does not. For instance:

```
relative(abs, "c:/Users/log.txt") == "..\\..\\Windows\\notepad.exe"
```

Comparing Paths

Two paths can be compared using the functions in the following table.

Function	Description
`equivalent(p1,p2)`	Two-way comparison that returns `true` if both `path`s refer to the same location after resolving relative paths, as well as all hard and symbolic links. Fails if either `path` does not exist.
`p1.compare(p2)` `p1.compare(str)`	Three-way lexical comparison method (returns an `int`). Hard or symbolic links are not resolved: comparison happens purely by three-way string comparison (as if by `basic_string::compare()`) on native pathname elements. Concretely, the result is determined as follows: • If the paths have different root names, return the result of comparing these. • Otherwise, return a negative value if `p1` does not have a root directory and `p2` does, and a positive value if `p1` has a root directory and `p2` does not. • Otherwise, apply element-wise three-way string comparison on all path elements, as if iterating simultaneously over the `relative_path()` parts of both `path`s and applying `native()` on each element before comparing. Return the first nonzero value. If all corresponding elements compare equal, return a negative value if `p1` has less elements, a positive one if `p2` has more elements, and zero otherwise.
`==, !=, <, <=, >, >=`	Two-way lexical comparison operators. Equivalent in semantics to `compare()`.

File Status

A `file_status` object stores two properties: a file type and file permissions. You can obtain a file's `file_status` either using `std::filesystem::status(path)` or from a `directory_entry` during directory listing (as explained later). For a symlink, `status()` returns the `file_status` of the link's target. `symlink_status()` is mostly equivalent to `status()`, except that for symlinks it retrieves the status of the symlink file itself.

Next to a set of constructors with optional type and permission values (in that order), a `file_status` only has `type()` and `permissions()` getters and setters. There are no other members.

■ **Caution** Modifying a `file_status` has no effect on the actual file. Normally, you therefore do not use the setters of a `file_status` (or its constructors for that matter). You normally obtain a `file_status` from a `[symlink_]status()` function and then use its getters to inspect a file's type or permissions. To alter permissions of a file, you can use the `permissions()` function discussed later. A file's type cannot be changed.

File Types

Some file systems offer more file types than just regular files, directories, and links. Table 5-6 provides an overview of possible file types and the related functionality.

Your first option to determine the type of a file is to use the `file_type` returned by the `type()` method of a `file_status`. Possible values of this scoped enumeration type are listed in the first column of Table 5-6 (implementations are allowed to define additional values).

Your second option are the functions listed in the second column of the table. They all exist both as non-member functions in the `std::filesystem` namespace and as member functions of a `directory_entry` (see the section on directory listing). The non-member functions accept either a `path` or a `file_status`.

■ **Tip** If you need to invoke multiple `is_file_type(my_path)` functions for a given `path`, it is more efficient first to obtain `status = file_status(my_path)` and then use `is_file_type(status)`.

Table 5-6. *Overview of Functions and `std::filesystem::file_type` Enumeration Values Used to Determine the Type of a File*

`file_type`	Functions	If the File…
`regular`	`is_regular_file()`	…is a regular file.
`directory`	`is_directory()`	…is a directory.
`symlink`	`is_symlink()`	…is a symbolic link.
`block`	`is_block_file()`	…is a block special file.
`character`	`is_character_file()`	…is a character special file.
`fifo`	`is_fifo()`	…is a FIFO file (also known as a pipe file).
`socket`	`is_socket()`	…is a socket file.
	`exists()`	…exists.
`not_found`		…does not exist.
	`is_other()`	…exists and is not a regular file, directory, or symbolic link.
`unknown`		…is an existing file for which the type could not be determined (for instance, due to lack of permissions).
`none`		…is a file for which type has not yet been determined (for instance, a default-constructed `file_status`), or for which an error has occurred while determining its type.
	`status_known()`	…is a file for which type has been determined. That is, a file whose type is different from `none` (and *not* different from `unknown`, as the function name suggests!).

File Permissions

Most file systems allow file permissions to be set on individual files and directories. The C++ file system library supports such permissions. It provides an enumeration called `std::filesystem::perms`, values of which can be bitwise combined to describe the file permissions of a file or directory. The following table lists all available enumeration values.

perms	Value	Description
`none`	`0`	No permissions are set for the file.

`owner_read`
`owner_write`
`owner_exec`
`owner_all`
	`0400`
	`0200`
	`0100`
	`0700`
		The file's owner has read, write, or execute permissions.
		`owner_all` = `owner_read` | `owner_write` | `owner_exec`.

`group_read`
`group_write`
`group_exec`
`group_all`
	`040`
	`020`
	`010`
	`070`
		Users in the file's user group have read, write, or execute permissions.
		`group_all` = `group_read` | `group_write` | `group_exec`.

`others_read`
`others_write`
`others_exec`
`others_all`
	`04`
	`02`
	`01`
	`07`
		Other users have read, write, or execute permissions.
		`others_all` = `others_read` | `others_write` | `others_exec`.

`all`	`0777`	Equivalent to `owner_all` | `group_all` | `others_all`.
`set_uid` `set_gid`	`04000` `02000`	Set user ID or group ID on execution. Useful to set on executable files. When a user executes a file which has the `set_uid` or `set_gid` permission flags set, the privileges of the user are temporarily elevated to the same privileges of the executable's user or group.
`sticky_bit`	`01000`	The meaning of this permission is operating system dependent. On Linux, e.g., files in a directory with the `sticky_bit` permission flag set can only be deleted or renamed by the file's owner, the directory's owner, or by the root user. If the `sticky_bit` permission is not set on the directory, any user with write and execute permissions on the directory can delete or rename any file in that directory.
`mask`	`07777`	Equivalent to `all` | `set_uid` | `set_gid` | `sticky_bit`.
`unknown`	`0xFFFF`	File permissions are unknown.

On Unix-like systems, the `read`, `write`, and `exec` permissions have the following semantics for directories:

- `read` grants the ability to list the names of the files present in the directory. To access the files themselves, or any meta-information beyond their names, you need `exec` permission.
- `write` grants the ability to create, delete, and rename files within the directory.
- `exec` grants the ability to *search* the directory—that is, to access the contents and meta-information (such as file type, size, permissions, etc.) of a file with a known name. To list the filenames inside a directory, the `read` permission remains required.

■ **Note** Not all implementations support all file permissions. Visual Studio's implementation, for instance, essentially only distinguishes between read-only files (none of the `write` bits are set) and files that are not read-only (the `all` bits are set). There's no distinction between owner, group, or others.

The `std::filesystem::permissions()` function can be used to modify the permissions for a given file:

```
void permissions(const path& path, perms permissions
                 [, perm_options o = perm_options::replace]);
```

The `std::filesystem::perm_options` parameter specifies how the permissions should be applied to the given file. It is an enumeration with the following values:

`perm_options`	Description
`replace`	The given permissions replace the current file's permissions. This is the default behavior.
`add`	The given permissions are bitwise OR'ed with the current file's permissions.
`remove`	The complement of the given permissions is bitwise AND'ed with the current file's permissions.
`nofollow`	By default the permissions of the file a symbolic link points to are modified. If the `nofollow` option is specified, however, the permissions of the symbolic link file itself are modified.

Creating, Copying, Deleting, and Renaming

Function	Description
`create_directory(p)` `create_directory(p,from)`	Creates a directory at the given `path` `p`. If a second `path` is provided, certain attributes (such as file permissions) are copied from the existing directory it must refer to. Which attributes are copied depends on the operating system. Returns `true` if a new directory was created, `false` otherwise. If the target directory already exists, `false` is returned without signaling failure.
`create_directories(p)`	Invokes `create_directory()` for any nonexistent directory in the given path. Returns `true` if one or more new directories were created, `false` otherwise.

`copy(from,to)`
`copy(from,to,options)`
`copy_file(from,to)`
`copy_file(from,to,opts)`
	`copy()` copies a regular file, directory, or symlink; `copy_file()` only copies a regular file. Fails if `paths` to wrong file types are provided, or if `from` and `to` are equivalent paths. The exact behavior is controlled by the chosen `copy_options` flags: see Table 5-7. The behavior of `copy()`/`copy_file()` is undefined if more than one flag of the same option group (the grouped rows in Table 5-7) is set.

`remove(path)`	Deletes a file, symlink, or empty directory. Fails if `path` corresponds to a nonempty directory. Symlinks are not followed: i.e., the symlink file itself is deleted, not its target. Returns `true` if the file existed and was removed; `false` otherwise.
`remove_all(path)`	Recursively deletes the contents of `path` if it is a directory as if by invoking `remove()`, followed by `remove(path)`.
`rename(from,to)`	Moves or renames a file. If `to` is the `path` of an existing file, that file gets deleted and replaced. `from` and `to` must then either both refer to a directory file or both refer to a nondirectory file. Does nothing if `from` and `to` are equivalent `paths`. Fails if `to` ends with dot or dot dot, or names a nonexistent directory ending with a directory separator. Symlinks are not followed: if `from` is a symlink, it is renamed and not its target. If `to` is an existing symlink, it itself is replaced, and not its target.

Table 5-7. *`std::filesystem::copy_options` Bitwise-Combinable Enumeration Values to Control `copy()` and `copy_file()` Operations*

`copy_options`	Description
`copy_options` to control `copy()`'s or `copy_file()`'s behavior with existing targets	
`none`	Fail and report an error. (Default)
`skip_existing`	Do not overwrite existing targets.
`overwrite_existing`	Always overwrite existing targets.
`update_existing`	Only overwrite an existing target if the new file is more recent.
`copy_options` to control `copy()`'s behavior with subdirectories	
`none`	Skip subdirectories. (Default)
`recursive`	Recursively copy subdirectories.
`copy_options` to control `copy()`'s behavior with symbolic links	
`none`	Follow symlinks, and copy the files they point to. (Default)
`copy_symlinks`	Copy symlinks as symlinks.
`skip_symlinks`	Do not copy symlinks (neither as symlinks nor as the files they point to).
`copy_options` to control `copy()`'s behavior	
`none`	Copy all content. (Default)
`directories_only`	Only copy directory structure. Do not copy nondirectory files.
`create_symlinks`	Instead of copying files, create symlinks to the source files. The source path shall be an absolute path unless the destination path is in the current directory.
`create_hard_links`	Instead of copying files, create hard links to the source files.

File Sizes and Free Space

You can query the size of a file with `std::filesystem::file_size(const path&)`. For a regular file, it returns the current file size in bytes (as a value of type `std::uintmax_t`, an alias for the largest integer type supported by your platform). Symlinks are always followed. For a file of any other type, and most notably for a directory, the result is implementation-defined. Returns `-1` on failure.

You can change the size of a file with `std::filesystem::resize_file(const path&, uintmax_t)`. If you increase the file size, the file is appended with zeros; otherwise the file is truncated to the smaller size.

`std::filesystem::space(const path&)` can be used to get information about the file system containing the given path. It returns a `std::filesystem::space_info` structure with the following `uintmax_t` members:

- `capacity`: The total size in bytes of the file system
- `free`: The number of free bytes on the file system
- `available`: The number of free bytes available for a nonprivileged process (less or equal to `free`)

Any `space_info` member that could not be determined is set to `-1`.

Directory Listing

Iterating over all files in a directory is made easy using a `directory_iterator`, as the following example shows:

```
// Print the names of all regular files in the current directory
std::filesystem::directory_iterator begin("."), end;
for (auto iter = begin; iter != end; ++iter)
   if (iter->is_regular_file())
      std::cout << iter->path().filename() << '\n';
```

The `directory_iterator` type also has range-based `for` loop support (by having `begin(iter)` return `iter`, and `end(iter)` a default-constructed iterator):

```
for (auto& entry : std::filesystem::directory_iterator("."))
  if (entry.is_regular_file())
     std::cout << entry.path().filename() << '\n';
```

To traverse a directory structure recursively, you can similarly use a `recursive_directory_iterator`:

```
for (auto& entry : std::filesystem::recursive_directory_iterator("."))
  if (entry.is_regular_file())
     std::cout << entry.path().filename() << '\n';
```

A directory iterator that is not an end iterator points to a `directory_entry`. Next to the usual copy and move members, and an (explicit) constructor from a given `path`, a `directory_entry` provides the following members:

Method	Description
`refresh()`	Refreshes all file attributes cached within the `directory_entry`. Any other accessor functions typically return values that were cached during directory listing, or when the `directory_entry` was last refreshed.
`path()` `operator const path&()`	Returns the `path` the `directory_entry` refers to.
`assign(p)`	Assigns a new `path p` to the `directory_entry` and invokes `refresh()`.
`replace_filename(p)`	Replaces the filename of the path of the `directory_entry` as if by `assign(path().replace_filename(p))`. This does not make changes to the underlying file system.

`file_size()`
`hard_link_count()`
`last_write_time()`
`status()`
`symlink_status()`
	Returns properties of the entry's file as if by invoking the non-member function with the same name on `path()`.

`is_`*`file_type`*`()` `exists()`	A set of members with the same name and semantics as the non-member functions described in Table 5-6 (with one exception: there is no `status_known()` member).

Both `directory_iterator` and `recursive_directory_iterator` have constructors accepting a `directory_options` as their second argument, which is an enumeration with the following values that can be bitwise combined:

directory_options	Description
`none`	The default behavior: neither `follow_directory_symlink` nor `skip_permission_denied` is set.
`follow_directory_symlink`	Symbolic links to directories are followed during iteration. This option only impacts subdirectories encountered during recursive iteration (and therefore has no effect for a `directory_iterator`). If the initial directory is a symlink, this link is always followed.
`skip_permission_denied`	Do not report an error when iteration fails to enter a directory due to lack of permissions. This option affects both the initial directory and all recursive subdirectories.

For both iterator types, the iteration order within a directory is unspecified. Special files "`.`" and "`..`" are always skipped. It is unspecified whether files that are deleted or added during an iteration still appear in the iteration or not.

A `recursive_directory_iterator` traverses a directory structure depth-first. More concretely: if an iterator is incremented while it points to a directory, the iterator starts listing its contents, recursively, before returning to the iteration of the parent directory. It has the following additional methods:

Method	Description
`depth()`	Returns the current recursion depth (zero initially, and incremented each time the iterator enters a new subdirectory)
`pop()`	Abandons the iteration of the current directory and returns to the iteration of the files in the parent directory
`disable_recursion_pending()`	Prevents the `recursive_directory_iterator` from entering the current file, should that be a directory. Resets each time the iterator is incremented
`recursion_pending()`	Checks whether `disable_recursion_pending()` has been called for the current file

C-Style File Utilities

`<cstdio>`

Prior to C++17, the following C-style functions in the `<cstdio>` header were the only portable file utilities available in the Standard Library:

Function	Description
`int remove(filename)`	Deletes the file with the given filename. Returns `0` on success. It is implementation dependent whether `errno` (see Chapter 8) is set when there is an error.
`int rename(old, new)`	The file named `old` is renamed to `new`. If supported, files may be moved to a different path as well. Returns `0` on success. It is implementation dependent whether `errno` (see Chapter 8) is set when there is an error.
`FILE* tmpfile()`	Opens a newly created file with a generated unique name for binary output. The returned `FILE*` pointer can be used, for instance, with the C-style I/O functions briefly discussed in the next section. The temporary file is automatically deleted when it is closed (for instance, using `std::fclose(FILE*)`). Returns `nullptr` when the file could not be created.
`char* tmpnam(char*)`	Creates a unique, then nonexisting filename. If a `char*` argument is provided, the result is stored in the buffer pointed to by this pointer and the same pointer is returned as well. The provided `char*` buffer must be at least `L_tmpnam` bytes long. If the argument is `nullptr`, a pointer to an internal static buffer is returned in which the filename is put. Returns `nullptr` if no filename could be generated.

■ **Caution** `tmpnam()` is not required to be thread-safe. It also does not create the file, so it is always possible that another process creates a file with the same name before your process does so. `tmpfile()` does not have these problems, but is incompatible with C++ file streams, nor can it be used to create directories. Native APIs (such as POSIX or the Windows API) typically provide better primitives for creating temporary files. Third-party libraries such as Boost.Filesystem and Qt offer suitable portable solutions.

C-Style Output and Input <cstdio>

Next to file utilities, the `<cstdio>` header also defines functions for character-based I/O (`getc()`, `putc()`, etc.) and formatted I/O (`printf()`, `scanf()`, etc.). All the C-style I/O functionality is subsumed by the type-safe C++ streams, which also have better-defined, portable error handling.[3] This section does discuss the `std::printf()` and `std::scanf()` families of functions, though, and only these, because they remain more convenient at times than C++ streams due to their compact formatting syntax.

std::printf() Family

The following `printf()` family of functions is defined in `<cstdio>`:

```
std::printf(const char* format, ...)
std::fprintf(FILE* file, const char* format, ...)
std::snprintf(char* buffer, size_t bufferSize, const char* format, ...)
std::sprintf(char* buffer, const char* format, ...)
```

They write formatted output to, respectively, standard output, a file, a buffer of given size, or a buffer and return the number of characters written out. The last one, `sprintf()`, is less safe than `snprintf()`. They all have a variable number of arguments following the `format` string. There are also versions prefixed with a `v` that accept a `va_list` for the arguments: for example, `vprintf(const char* format, va_list)`. For the first three, wide-character versions are provided as well: `(v)wprintf()`, `(v)fwprintf()`, and `(v)swprintf()`.

The given `format` string controls how the output is formatted. All characters of this `format` string are written out as is, except sequences that start with a `%`. The basic syntax for a formatting option is `%` followed by a *conversion specifier*. This tells `printf()` how to interpret the next value in the variable-length list of arguments. The arguments passed to `printf()` must be in the same order as the `%` directives in `format`. Table 5-8 explains the available conversion specifiers. The expected argument types listed are for the case where no length modifier is used (discussed later).

[3] Some library implementations use `errno` (see Chapter 8) to report errors for C-style I/O functions, including the `printf()` and `scanf()` functions: consult your library documentation to confirm.

Table 5-8. *Available Conversion Specifiers for `printf()`-Like Functions*

Specifier	Description
`d, i`	A `signed int` argument converted to decimal representation `[-]ddd`.
`o, u, x, X`	An `unsigned int` argument converted to an octal (`o`), decimal (`u`), or hexadecimal representation, the latter with either lowercase (`x`) or uppercase digits (`X`).
`f, F`	A `double` argument converted to a decimal notation in the style `[-]ddd.dd` (with lowercase or, respectively, uppercase letters used for infinity and NaN values).
`e, E`	A `double` argument converted to scientific notation: i.e., `[-]d.dde±dd` or `[-]d.ddE±dd` (again with lowercase/uppercase letters for special values).
`g, G`	A `double` argument converted as if with `f/F` or `e/E`, whichever is more compact for the given value and precision. `e/E` is only used if the exponent is greater than or equal to the precision, or less than -4.
`a, A`	A `double` argument converted to hexadecimal format: `[-]0xh.hhhp±d` or `[-]0Xh.hhhP±d` (infinity and NaN values are printed as with `f, F`).
`c`	An `int` argument converted to a single `unsigned char`.
`s`	The argument is a pointer to a `char` array. The precision specifies the maximum number of bytes to output. If no precision is given, writes everything until the null terminator. Note: *Do not* pass a `std::string` object as is as argument for a `%s` modifier!
`p`	The argument is interpreted as a `void` pointer, and the pointer is converted to an implementation-dependent format.
`n`	The argument is a pointer to a `signed int` that receives the number of characters written out so far by this call to `printf()`.
`%`	Outputs a `%` character. No corresponding argument must be passed.

■ **Caution** The C-style I/O functions are not type-safe. If your conversion specifier says to interpret an argument value as a `double`, then that argument must be a true `double` (and not, for instance, a `float` or an integer). It will compile and run if a wrong type is passed, but this rarely ends well. This also means you should never pass a C++ `std::string` as is as an argument for a string conversion specifier: instead, use `c_str()` as shown in the following example.

The following example prints the lyrics of the traditional American folk song "99 Bottles of Beer":

```
std::string bottles = "bottles of beer";
char on_wall[99];
for (int i = 99; i > 0; --i) {
   snprintf(on_wall, sizeof(on_wall), "%s on the wall", bottles.c_str());
   printf("%d %s, %d %s.\n", i, on_wall, i, bottles.c_str());
   printf("Take one down, pass it around, %d %s.\n", i-1, on_wall);
}
```

The formatting options are much more powerful than the basic conversions discussed so far. The full syntax of the % directive is as follows (the < and > tags are included for exposition only):

```
%<flags><width><precision><length_modifier><conversion>
```

with

- `<flags>`: Zero or more flags that change the meaning of the conversion specifier. See Table 5-9.
- `<width>`: Optional *minimum* field width (truncation is never done: only padding). Padding is applied if the converted value has fewer characters than the specified width. By default, spaces are used for padding. `<width>` can be either a non-negative integer or *, which means to take the width from an integer argument from the argument list. This width has to precede the value to be formatted.
- `<precision>`: A dot followed by an optional non-negative integer (`0` is assumed if not specified), or a *, which again means to take the precision from an integer argument from the argument list. The precision is optional and determines the following:
  - The maximum number of bytes for `s`. By default, a zero-terminated character array is expected.
  - The minimum number of digits to output for all integer conversion specifiers (`d`, `i`, `o`, `u`, `x`, and `X`). Default: 1.
  - The number of digits to output after the decimal point for most floating-point conversion specifiers (`a`, `A`, `e`, `E`, `f`, and `F`). If not specified, the default precision is 6.
  - The maximum number of significant digits for `g` and `G`. The default is again 6.

- `<length_modifier>`: An optional modifier that alters the type of the argument to be passed. Table 5-10 gives an overview of all supported modifiers for numeric conversions. For character and strings (`c` and `s` conversion specifiers, respectively), the `l` length modifier (note: this is the letter `l`) changes the expected input type from `int` and `char*` to `wint_t` and `wchar_t*`, respectively.[4]
- `<conversion>`: The only required component, which specifies the conversion to apply to the argument. (See Table 5-8.)

Table 5-9. *Available Flags*

Flag	Description
-	Left justifies the output. By default, output is right justified.
+	Always outputs the sign of a number, even for positive numbers.
space-character	Prefixes the output with a space if the number to output is non-negative or results in no characters. Ignored if + is also specified.
#	Outputs a so-called *alternative form*. For `x` and `X`, the result is prefixed with `0x` or `0X` if the number is not zero. For all floating-point specifiers (`a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G`), the output always contains a decimal point character. For `g` and `G`, trailing zeros are not removed. For `o`, precision is increased such that the first digit to output is a zero.
0	For all integer and floating-point conversion specifiers (`d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G`), padding is done with zeros instead of spaces. Ignored if - is specified as well, or for all integer specifiers in combination with a precision.

The modifiers in Table 5-10 determine the type of the inputs that must be passed as indicated. `std::intmax_t` and `uintmax_t` are defined in `<cstdint>` (see Chapter 1), and `size_t` and `ptrdiff_t` are defined in `<cstddef>`. Note also that the `long` and `long long` modifiers use the letter `l`, and not the number `1`.

[4] `wint_t` is defined in `<cwchar>` and is an alias for an integral type large enough to hold any wide character (`wchar_t` value) and at least one value that is not a valid wide character (`WEOF`).

Table 5-10. *Length Modifiers for All Numeric Conversion Specifiers*

Modifier	d, i	o, u, x, X	n	a, A, e, E, f, F, g, G
(none)	int	unsigned int	int*	double
hh	char	unsigned char	char*	
h	short	unsigned short	short*	
l	long	unsigned long	long*	
ll	long long	unsigned long long	long long*	
j	intmax_t	uintmax_t	intmax_t*	
z	size_t	size_t	size_t*	
t	ptrdiff_t	ptrdiff_t	ptrdiff_t*	
L				long double

Example

```
int i = 123;
std::printf("i: '%+10d'\n", i);                // i: '      +123'

long double d = 31.415;
int prec = 4;       /* precision */
std::printf("d: %.4Lf = %.*Le\n", d, prec, d); // d: 31.4150 = 3.1415e+01
```

std::scanf() Family

The following `scanf()` family of functions is defined in `<cstdio>`:

```
std::scanf(const char* format, ...)
std::fscanf(FILE* file, const char* format, ...)
std::sscanf(const char* buffer, const char* format, ...)
```

They read, respectively, from standard input, a file, or a buffer. In addition to these functions, which have a variable number of arguments following the `format` string, there are also versions whose names are prefixed with `v` and that accept a `va_list` for the arguments: for example, `vscanf(const char* format, va_list)`. Wide-character versions are provided as well: `(v)wscanf()`, `(v)fwscanf()`, and `(v)swscanf()`.

They all read formatted data based on a given `format` string. The `scanf()` formatting grammar used is similar to that of `printf()`, seen earlier. All characters in the format string are simply used to compare with the input, except sequences that start with a `%`. These `%` directives result in values being parsed and stored in the location pointed to by

the function's arguments, in order. The basic syntax is a % sign followed by one of the conversion specifiers from Table 5-11. The last column shows the argument type in case no length modifiers are specified (see Table 5-12).

Table 5-11. *Available Conversion Specifiers for* `scanf()`*-Like Functions*

Specifier	Matches…	Argument
`d`	Optionally signed decimal integer.	`int*`
`i`	Optionally signed integer whose base is determined from the integer's prefix: decimal by default, but octal if it starts with `0` and hexadecimal if it starts with `0x` or `0X`.	`int*`
`o` / `u` / `x`, `X`	Optionally signed octal/decimal/hexadecimal integer. Hexadecimal digits are optionally preceded by `0x` or `0X`.	`unsigned int*`
`a`, `A`, `e`, `E`, `f`, `F`, `g`, `G`	Optionally signed floating-point number, infinity, or NaN. All eight specifiers are completely equivalent: e.g., they all parse scientific notation as well.	`float*`
`c`	A character sequence whose length is specified by the field width, or of length one if no width is specified.	`char*`
`s`	A sequence of nonwhitespace characters.	`char*`
`[...]`	A nonempty character sequence from a set of expected characters. The set is specified between square brackets, e.g., `[abc]`. To match all characters except those in a set, use `[^abc]`.	`char*`
`p`	An implementation-dependent sequence of characters as produced by `%p` with `printf()`.	`void**`
`n`	Does not extract/parse any input. The argument receives the number of characters read from the input stream so far.	`int*`
`%`	A % character.	/

For all directives except those with conversion specifier `c`, `s`, or `[...]`, any whitespace characters are skipped until the first nonwhitespace one. Parsing stops when the end of the input string is reached, when a stream input error occurs, or when a parsing error occurs. The return value equals the number of assigned values or `EOF` if an input failure occurred before starting the first conversion. The number of assigned values will be less than the number of directives if the end of the stream is reached or a parsing error occurs: for example, zero if this occurs during the first conversion.

The full syntax of the % directive is as follows (all < and > tags are for exposition only):

```
%<*><width><length_modifier><conversion>
```

with

- <*>: An optional * sign that causes scanf() to parse the data from the input without storing it in any of the arguments.
- <width>: Optional maximum field width in characters.
- <length_modifier>: Optional length modifier; see Table 5-12. When applied to a c, s, or [...] specifier, the l (letter l) modifies the required input type from char* to wchar_t*.
- <conversion>: Required. Specifies the conversion to apply; see Table 5-11.

Table 5-12. *Available Length Modifiers for the Numeric Conversion Specifiers of scanf()-Like Functions*

Modifier	d, i	o, u, x, X	n	a, A, e, E, f, F, g, G
(none)	int*	unsigned int*	int*	float*
hh	char*	unsigned char*	char*	
h	short*	unsigned short*	short*	
l	long*	unsigned long*	long*	double*
ll	long long*	unsigned long long*	long long*	
j	intmax_t*	uintmax_t*	intmax_t*	
z	size_t*	size_t*	size_t*	
t	ptrdiff_t*	ptrdiff_t*	ptrdiff_t*	
L				long double*

The only nonobvious difference between Table 5-12 for scanf() functions and the corresponding Table 5-10 for printf() functions is that by default, floating-point arguments must point to a float and not a double.

Example

```
std::string s = "i: +123; d: -2.34E-3; chars: abcdef";
int i = 0; double d = 0.0; char chars[4] = { 0 };
std::sscanf(s.c_str(), "i: %i; d: %lE; chars: %[abc]", &i, &d, chars);
std::printf("i: %+i; d: %.2lE; chars: %s", i, d, chars);
```

CHAPTER 6

Characters and Strings

Strings

<string>

C++ supports *C-style strings*—plain C-style character arrays terminated with a null character (`'\0'`, with ASCII code zero). The C Standard Library offers various headers with functions to manipulate such strings. Working directly with C-style strings, however, has serious drawbacks. Analogous to working with plain arrays (compared to working with, say, `std::vector`), a C-style string doesn't know its own size, and it is your job to allocate the required memory. It also falls to you to ensure these C-style strings are, and always remain, properly null-terminated. We therefore recommend you do not work directly with C-style strings, and use the container-like abstractions offered by the C++ Standard Library instead.

The `<string>` header defines four different string types, each for a different `char`-like type:

	String Type	Characters	Typical Character Size
Narrow strings	`std::string`	`char`	8 bit
Wide strings	`std::wstring`	`wchar_t`	16 or 32 bit
UTF-16 strings	`std::u16string`	`char16_t`	16 bit
UTF-32 strings	`std::u32string`	`char32_t`	32 bit

The names in the first column are purely indicative, because strings are completely agnostic about the character encoding used for the `char`-like items—or *code units,* as they are technically called—they contain. Narrow strings, for example, may be used to store ASCII strings, as well as strings encoded using UTF-8 or DBCS.

In this section, we will mostly use `std::string`. But everything applies equally well to all other types. The locale and regular expression functionalities discussed in subsequent sections are, unless otherwise noted, only required to be implemented for narrow and wide strings.

All four string types are instantiations of the same class template, `std::basic_string<CharT>`. A `basic_string<CharT>` is essentially a `vector<CharT>` with extra functions and overloads either to facilitate common string operations or for compatibility with C-style strings (`const CharT*`). All members of `vector` are provided for strings as

P. Van Weert and M. Gregoire, *C++17 Standard Library Quick Reference*,
https://doi.org/10.1007/978-1-4842-4923-9_6

well, except for the emplacement functions (which are of little use for characters). This implies that, unlike in other mainstream languages such as .NET, Python, and Java, strings in C++ are mutable. It also means, for example, that strings can readily be used with all algorithms seen in Chapter 4:

```
std::string s = "Strings be fun";
s.reserve(20);
s.push_back('!');                                  // "Strings be fun!"
const auto found = std::find(cbegin(s), cend(s), 'b');
s[found - s.cbegin()] = 'r';                       // "Strings re fun!"
s.insert(found, 'a');                              // "Strings are fun!"
```

The remainder of this section focuses on the functionality that strings add compared to `vectors`. For the functions that strings have in common with `vector`, we refer to Chapter 3. One thing to note is that string-specific functions and overloads are mostly index-based rather than iterator-based. The last three lines in the previous example, for instance, may be written more conveniently as

```
const size_t found = s.find('b');
s[found] = 'r';
s.insert(found, "a");  // (no index-based single-character insert exists)
```

or

```
const size_t found = s.find("be");
s.replace(found, 2, "are");       // 2 = number of characters to replace
```

The equivalent of the `end()` iterator when working with string indices is `basic_string::npos`. This constant is consistently used to represent half open-ended ranges (i.e., to denote "until the end of the string"), and, as you see next, as the "not found" return value for `find()`-like functions.

Searching in Strings

Strings offer six member functions to search for substrings or characters: `find()` and `rfind()`, `find_first_of()` and `find_last_of()`, and `find_first_not_of()` and `find_last_not_of()`. These always come in pairs: one to search from front to back, and one to search from back to front. All also have the same five overloads of the following form:

```
size_t find(char c, size_t pos = 0) const;                  // single char
size_t find(const string& str, size_t pos = 0) const noexcept;   // string
size_t find(const char* s, size_t pos = 0) const;              // C-string
size_t find(const char* s, size_t pos, size_t n) const;     // char buffer
size_t find(string_view sv, size_t pos = 0) const;          // string view
```

The mostly optional pos parameter is the index at which the search should start. For the functions searching backward, the default value for pos is npos.

The pattern to search for is either a single character or a string, with the latter represented as a C++ string, a null-terminated C-string, a character buffer of which the first n values are used, or a string view (the C++17 string_view type is discussed later in this chapter). The (r)find() functions search for an occurrence of the full pattern, and the find_xxx_of() / find_xxx_not_of() family of functions search for any single character that occurs/does not occur in the pattern. The result is the index of the (start of the) first occurrence starting from either the beginning or end, or npos if no match is found.

■ **Tip** Starting with C++17, the <algorithm> header offers more efficient algorithms for finding substrings. These are explained in Chapter 4.

Modifying Strings

To modify a string, you can use all members known already from vector, including erase(), clear(), push_back(), and so on (see Chapter 3). Additional functions or functions with string-specific overloads are assign(), insert(), append(), +=, and replace(). Their behavior should be obvious; only replace() may need some explanation. First though, let's introduce the multitude of useful overloads these five functions have. These are generally of this form:

```
string& func([Position,] const string& str);  // C++ string
string& func([Position,] const string& str,   // substring of C++ string
             size_t substringPos, size_t substringLen = npos);
string& func([Position,] const char* str);    // null-terminated C-string
string& func([Position,] const char* str, size_t n); // buffer of n chars
string& func([Position,] string_view sv);     // string view (see later)
string& func([Position,] size_t n, char c);   // fill
string& func([Position,] InputIterator first, InputIterator last); // (*)
string& func([Position,] initializer_list<char> il);               // (*)
```

For moving a string, assign(string&&) is defined as well. Because the += operator inherently only has a single parameter, naturally only the C++ string, C-style string, and initializer-list overloads are possible.

Analogous to their vector counterparts, for insert() the overloads marked with (*) return an iterator rather than a string. Likely for the same reason, the insert() function has these two additional overloads:

```
iterator insert(const_iterator position, size_t n, char c);   // fill
iterator insert(const_iterator position, char c);      // single char
```

Only insert() and replace() need a Position. For insert(), this is usually an index (a size_t), except for the last two overloads, where it is an iterator (analogous again to vector::insert()). For replace(), the Position is a range, specified either using two

const_iterators (not available for the substring overload) or using a start index and a length (not for the last two overloads).

In other words, replace() does not, as you may expect, replace occurrences of a given character or string with another. Instead, it replaces a specified subrange with a new sequence—a string, substring, fill pattern, and so on—possibly of different length. You saw an example of its use earlier (2 is the length of the replaced range):

```
s.replace(s.find("be"), 2, "are");
```

To replace all occurrences of substrings or given patterns, you can use regular expressions and the std::regex_replace() function explained later in this chapter. To replace individual characters, the generic std::replace() and replace_if() algorithms from Chapter 4 are an option as well.

A final modifying function with a noteworthy difference from its vector counterpart is erase(): in addition to the two iterator-based overloads, it has one that works with indices. Use it to erase the tail or a subrange or, if you like, to clear() it:

```
string& erase(size_t pos = 0, size_t len = npos);
```

Constructing Strings

In addition to the default constructor, which creates an empty string, the constructor has the same seven overloads as the functions in the previous subsection, plus of course one for string&&. (Like other containers, all string constructors have an optional argument for custom allocators as well, but this is for advanced use only.)

As of C++14, basic_string objects of various character types can also be constructed from corresponding string literals by appending the suffix s. This literal operator is defined in the std::literals::string_literals namespace:

```
using namespace std::literals::string_literals;
auto a = "a is a const char*";
auto b = "b is a std::string"s;
auto c = std::pair(3u, L"c is a pair<unsigned, wstring>"s); // <utility>
```

String Length

To get a string's length, you can use either the typical container member size() or its string-specific alias length(). Both return the number of char-like elements the string contains. Take care, though: C++ strings are agnostic on the character encoding used, so their length equals what is technically called the number of *code units*, which may be larger than the number of *code points* or *characters*. Well-known encodings where not all characters are represented as a single code unit are the variable-length Unicode encodings UTF-8 and UTF-16:

```
std::string s(u8"字符串"); // UTF-8 encoding of Chinese word for "string"
std::cout << s.length();   // Length: 9 code units!
```

One way to get the number of code points is to convert to a UTF-32 encoded string first, using the character-encoding conversion facilities introduced later in this chapter.

Copying (Sub)Strings

Another `vector` function (next to `size()`) that has a string-specific alias is `data()`, with its equivalent `c_str()`. Both return a `const` pointer to the internal character array (without copying). To copy the string to a C-style string instead, use `copy()`:

```
size_t copy(char* out, size_t len, size_type pos = 0) const;
```

This copies `len` `char` values starting at `pos` to `out`. That is, it may be used to copy a substring as well. To create a substring as a C++ string, use `substr()`:

```
string substr(size_t pos = 0, size_t len = npos) const;
```

Comparing Strings

Strings may be compared lexicographically with other C++ strings or C-style strings using either the non-member comparison operators (`==`, `<`, `>=`, etc.) or their `compare()` member. The latter is a so-called *three-way comparison* function and has several overloads as listed next. The overloads with a `std::string` also exist for a `string_view` argument (`string_view` is explained in the next section):

```
int compare(const string& str) const noexcept;
int compare(size_type pos1, size_type n1, const string& str
            [, size_type pos2, size_type n2 = npos]) const;
int compare(const char* s) const;
int compare(size_type pos1, size_type n1, const char* s
            [, size_type n2]) const;
```

`pos1`/`pos2` is the position in the first/second string where the comparison should start, and `n1`/`n2` is the number of characters to compare from the first/second string. The return value is zero if both strings are equal, and a negative/positive number if the first string is less/greater than the second.

■ **Caution** It is a common mistake to confuse a three-way comparison function for a function that checks for equality, like in the following snippet:

```
if (s1.compare(s2)) /* strings are equal ... */
```

This has the opposite effect as intended. If two strings are equal, `compare()` evaluates to zero, which becomes `false` if converted to a Boolean.

String Views C++17 <string_view>

`std::string_view` acts as a non-owning, read-only view on a sequence of characters, functionally nearly equivalent to a reference of type `const std::string&`. The main difference is that when using `std::string_view`, you never inadvertently copy the underlying character sequence. Consider this function that extracts the extension of a given filename (for simplicity, assume there always is one):

```
std::string extension(const std::string& fileName) {
   return fileName.substr(fileName.rfind('.') + 1);
}
```

To evaluate `extension("my_pic.png")`, at least two character sequences are copied: first `"my_pic.png"` is copied into a temporary `std::string` object that binds with the `fileName` parameter, and then `"png"` is copied into the `std::string` object that is returned from the function. Of course, for this toy example the overhead of unnecessarily copying read-only input strings is entirely negligible, but in general this cost could be significant.

Traditionally, avoiding such overheads involved adding multiple overloads for string literals, C-style null-terminated strings, and so on. Not only does this mean more work, these overloads then do not have access to the convenient member functions that `std::string` offers either (such as `substr()` and `rfind()`).

Starting with C++17, you can instead simply write functions such as `extension()` as follows:

```
std::string_view extension(std::string_view fileName) {
   return fileName.substr(fileName.rfind('.') + 1);
}
```

Now `extension("my_pic.png")` no longer copies any character sequences. And this same function works for `std::string` or C-style `char*` pointers just as well.

■ **Note** `const std::string_view&` would be fine as parameter type as well. But since a `string_view` normally only holds a pointer and a size, copying `string_views` is cheap enough to pass them by value.

A `string_view` object only has `const` methods that allow you to *view* a string, nothing more. That is, you cannot modify the underlying character array through its interface. Still, because it may save gratuitous copying, at virtually no cost, `std::string_view` should generally be preferred over a reference of type

`const std::string&`. It is almost always a drop-in replacement for `const std::string&`, except for these small differences:

- `string_view` does not offer `c_str()`, only `data()`. Reason is that unlike a `string`, a `string_view` does not always point to a buffer that is null-terminated at `data()` + `size()`.
- Creating a `string` object from a `string_view` requires an explicit construction (to avoid inadvertent copies). The other way around, a `string` object does implicitly convert to `string_view` (thanks to the cast operator added in C++17).
- `string_view` offers `remove_prefix(`*n*`)` and `remove_suffix(`*n*`)` member functions to create new `string_views` for substrings with *n* characters removed from the start or end, respectively.

To create a `string_view` from a string literal, you can use the `""sv` literal operator defined in the `std::literals::string_view_literals` namespace. As always, both `literals` and `string_view_literals` are inline namespaces:

```
using namespace std::string_view_literals; // or std::literals, or...
auto my_sv_literal = "The best view comes after the hardest climb"sv;
```

■ **Caution** A `string_view` does not copy or take ownership of the underlying character array. It is your responsibility to ensure this character array outlives the `string_view`. Creating `string_views` for plain string literals is always safe.

The `<string_view>` header also defines `std::wstring_view`, `u16string_view`, and `u32string_view` for unmodifiable views on strings and character sequences consisting of characters of types other than `char`. As always, all four `string_view` types are aliases for instantiations of a `basic_string_view<CharT>` template.

Character Classification `<cctype>, <cwctype>`

The `<cctype>` and `<cwctype>` headers offer a series of functions to classify, respectively, `char` and `wchar_t` characters. These functions are `std::is`*class*`(int)` and `std::isw`*class*`(wint_t)`, where *class* equals one of the values in Table 6-1 (and `wint_t` is a type alias for an integer type). All functions return a nonzero `int` if the given character belongs to the class, or zero otherwise.

Table 6-1. *The 12 Standard Character Classes*

Class	Description
`cntrl`	Control characters: all non-`print` characters. Includes '`\0`', '`\t`', '`\n`', '`\r`', etc.
`print`	Printable characters: digits, letters, space, punctuation marks, etc.
`graph`	Characters with graphical representation: all `print` characters except ' '
`blank`	Whitespace characters that separate words on a line. At least ' ' and '`\t`'
`space`	Whitespace characters: at least all `blank` characters, '`\n`', '`\r`', '`\v`', and '`\f`'. Never `alpha` characters
`digit`	Decimal digits (`0`–`9`)
`xdigit`	Hexadecimal digits (`0`–`9`, `A`–`F`, `a`–`f`)
`alpha`	Letter characters. At least all `lowercase` and `uppercase` characters, and never any of the `cntrl`, `digit`, `punct`, and `space` characters
`lower`	Lowercase `alpha` letters (`a`–`z` for the default locale)
`upper`	Uppercase `alpha` letters (`A`–`Z` for the default locale)
`alnum`	Alphanumeric characters: union of all `alpha` and `digit` characters
`punct`	Punctuation marks (`! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~` for the default locale). Never a `space` or `alnum` character

The same headers also offer the `tolower()` / `toupper()` and `towlower()` / `towupper()` functions for converting between lowercase and uppercase characters. Characters are again represented using `int` and `wint_t`. If the conversion is undefined or impossible, these functions simply return their input value.

All character classification and transformation functions are defined only for `int`s / `wint_t` inputs that are representable as `unsigned char` / `wchar_t`, respectively, plus the value of the macro `EOF` / `WEOF`. The latter macros expand to the (negative) integer used to represent the end of a file or stream (EOF is short for *end-of-file*).

The exact behavior of all these functions depends on the active C locale. Locales are explained in detail later in this chapter, but essentially this means the active language and regional settings may result in different sets of characters to be considered letters, lowercase or uppercase, digits, whitespace, and so on. Table 6-1 lists all general properties of and relations between the different character classes and gives some examples for the default `"C"` locale.

■ **Note** In the "Localization" section later in this chapter, you see that the `<locale>` header also offers a list of overloads for `std::is`*class*`()` and `std::tolower()` / `toupper()` (all templated on the character type) that use a given C++ `locale` rather than the active `C` locale.

Character-Encoding Conversion `<locale>, <codecvt>`

■ **Note** Most of the functionality we explain in this section is now deprecated. (Concretely, the three conversion classes defined by the `<codecvt>` header (Table 6-3), as well as the `std::wstring_convert` and `wbuffer_convert` helper classes of `<locale>`, have been deprecated in C++17). Also, the support for these facilities in mainstream implementations was never very good to begin with. To manipulate Unicode-encoded text in a portable manner, we therefore recommend you use a third-party library instead. Viable candidates include the powerful ICU library or Boost.Locale.

A *character encoding* determines how *code points* (many but not all code points are characters) are represented as binary *code units*. Examples include ASCII (classical encoding with 7-bit code units), the fixed-length UCS-2 and UCS-4 encodings (16-bit and 32-bit code units, respectively), and the three main Unicode encodings: the fixed-length UTF-32 (using a single 32-bit code unit for each code point) and variable-length UTF-8 and UTF-16 encodings (representing each code point as one or more 8- or 16-bit code units, respectively; up to 4 units for UTF-8, and 2 for UTF-16). The details of Unicode and the various character encodings and conversions could fill a book; we explain here what you need to know in practice to convert between encodings.

The class template for objects that contain the low-level encoding-conversion logic is `std::codecvt<CharType1, CharType2, State>` (`cvt` is likely short for `converter`). It is defined in `<locale>` (as you see in the next section, this is actually a *locale facet*). The first two parameters are the C++ character types used to represent the code units of both encodings. For all standard instantiations, `CharType2` is `char`. `State` is an advanced parameter we do not explain further (all standard specializations use `std::mbstate_t` from `<cwchar>`).

The four `codecvt` specializations listed in Table 6-2 are defined in `<locale>`. Additionally, the `<codecvt>` header defines the three `std::codecvt` subclasses listed in Table 6-3.[1] For these, `CharT` corresponds to the `CharType1` parameter of the `codecvt` base class; as stated earlier, `CharType2` is always `char`.

[1] These classes have two more optional template parameters: a number specifying the largest code point to output without error and a `codecvt_mode` bitmask value (default `0`) with possible values `little_endian` (output encoding) and `consume_header` / `generate_header` (read/write initial BOM header to determine endianness).

Table 6-2. *Character-Encoding Conversion Classes Defined in <locale>*

`codecvt<char,char,mbstate_t>`	Identity conversion
`codecvt<char16_t,char,mbstate_t>`	Conversion between UTF-16 and UTF-8
`codecvt<char32_t,char,mbstate_t>`	Conversion between UTF-32 and UTF-8
`codecvt<wchar_t,char,mbstate_t>`	Conversion between native wide and narrow character encodings (implementation specific)

Table 6-3. *Character-Encoding Conversion Classes Defined in <codecvt>*

`codecvt_utf8<CharT>` `codecvt_utf16<CharT>`	Conversion between UCS-2 (for 16-bit `CharT`s) or UCS-4 (for 32-bit `CharT`s) and UTF-8/UTF-16. The UTF-16 string is represented using 8-bit `char`s as well, so this is intended for binary UTF-16 encodings.
`codecvt_utf8_utf16<CharT>`	Conversion between UTF-16 and UTF-8 (`CharT` must be at least 16-bit).

Although `codecvt` instances could in theory be used directly, it is far easier to use the `std::wstring_convert<CodecvtT, WCharT=wchar_t>` class from `<locale>`. This helper class facilitates conversions between `char` strings and strings of a (generally wider) character type `WCharT` in both directions. Despite its misleading (outdated) name, `wstring_convert` can also convert from and to, for example, `u16string`s or `u32string`s, not just `wstring`s. These members are provided:

Member	Description
(constructor)	Constructors exist that take a pointer to an existing `CodecvtT` (of which `wstring_convert` takes ownership) and an initial state (not discussed further). Both are optional. A final constructor accepts two error strings: one to be returned by `to_bytes()` upon failure and one by `from_bytes()` (the latter is optional).
`from_bytes()`	Converts either a single `char` or a string of `char`s (a C-style `char*` string, a `std::string`, or a sequence bounded by two `char*` pointers) to a `std::basic_string<WCharT>` and returns the result. Throws `std::range_error` upon failure, unless an error string was provided upon construction: in that case, this error string is returned.
`to_bytes()`	Opposite conversion from `WCharT` to `char`, with analogous overloads.
`converted()`	Returns the number of input characters processed by the last `from_bytes()` or `to_bytes()` conversion.
`state()`	Returns the current state (mostly `mbstate_t`: not discussed further).

Recall the following example from the section on `std::string` lengths:

```
std::string s(u8"字符串"); // UTF-8 encoding of Chinese word for "string"
std::cout << s.length();   // Length: 9 code units!
```

To convert this string to UTF-32, you would hope the following is possible:

```
using cvt = std::codecvt<char32_t,char,std::mbstate_t>;
std::wstring_convert<cvt, char32_t> convertor;
std::u32string s_u32 = convertor.from_bytes(s);
std::cout << s_u32.length(); // Length: 3 code units
```

Unfortunately, this does not compile. For the converter subclasses defined in `<codecvt>`, this would compile. But the destructor of the `codecvt` base class is `protected` (like all standard locale facets: discussed later), and the `wstring_convert` destructor calls it to delete the converter instance it owns. This design defect can be circumvented using a helper wrapper such as the following:

```
// Inherit all constructors and make a protected destructor public:
template <class BASE> class Deletable : public BASE {
  public:
    using BASE::BASE;
    ~Deletable() = default;
};
```

To make the code compile,[2] you then replace the first line with the following:

```
using cvt = Deletable<std::codecvt<char32_t,char,std::mbstate_t>>;
```

To use the potentially locale-specific variants of these converters (see the next section), use the following (other locale name besides "" may be used as well):

```
using cvt = Deletable<std::codecvt_byname<char32_t,char,std::mbstate_t>>;
std::wstring_convert<cvt, char32_t> convertor(new cvt(""));
```

A related class is `wbuffer_convert<CodecvtT, WCharT=wchar_t>`, which wraps a `basic_streambuf<char>` and makes it act as a `basic_streambuf<WCharT>` (stream buffers are very briefly explained in Chapter 5). A `wbuffer_convert` instance is constructed with an optional `basic_streambuf<char>*`, `CodecvtT*`, and state. Both the getter and setter for the wrapped buffer are called `rdbuf()`, and the current conversion state may be obtained using `state()`.

[2] At the time of writing, this example does not work in Visual Studio. It compiles after replacing `char32_t` with `__int32`, and `u32string` with `basic_string<__int32>`, but even then the results are wrong.

The following constructs a stream that accepts wide character strings, but writes it to an UTF-8 encoded file (needs <fstream>):

```
std::ofstream out("test.txt");    // char-based file output stream
std::wbuffer_convert<std::codecvt_utf8<wchar_t>> cvt(out.rdbuf());
std::wostream wout(&cvt);         // wchar_t output stream
wout << L"I am written as UTF-8, irrespective of the native wide format!";
```

Localization <locale>

Textual representations of dates, monetary values, and numbers are governed by regional and cultural conventions. To illustrate, the following three sentences are analogous but written using local currencies, numeric, and date formats:

```
In the U.S., John Doe has won $100,000.00 on the lottery on 3/14/2015.
In India, Ashok Kumar has won ₹ 1,00,000.00 on the lottery on 14-03-2015.
En France, Monsieur Brun a gagné 100.000,00 € à la loterie sur 14/3/2015.
```

In C++, all parameters and functionality related to processing text in a *locale-specific* manner—that is, adapted to local conditions—are contained in a `std::locale` object. These include not only formatting of numeric values and dates as just illustrated but also locale-specific sorting and conversions of strings.

Locale Names

Standard `locale` objects are constructed from a *locale name*:

```
std::locale(const char* locale_name);
std::locale(const std::string& locale_name);
```

These names commonly consist of a two-letter ISO-639 *language code* followed by a two-letter ISO-3166 *country code*. The precise format, however, is platform specific: on Windows, for instance, the name for the English-American locale is `"en-US"`, whereas on POSIX-based systems it is `"en_US"`. Most platforms support, or sometimes require, additional specifications such as region codes, character encodings, and so on. Consult your platform's documentation for a full list of supported locale names and options.

There are only two portable locale names, `""` and `"C"`:

- With `""`, you construct a `std::locale` with the user's preferred regional and language settings, taken from the program's execution environment (i.e., the operating system).
- The `"C"` locale denotes the *classic* or *neutral* locale, which is the standardized, portable locale that all C and C++ programs use by default.

Using the "C" locale, the earlier example sentence becomes

```
Anywhere, a C/C++ programmer may win 100000 on the lottary on 3/14/2015.
```

■ **Tip** When writing to a file intended to be read by computer programs (configuration files, numeric data output, etc.), it is highly recommended that you use the neutral "C" locale, to avoid problems during parsing. When displaying values to the user, you should consider using a locale based on the user's preferences ("").

The Global Locale

The active *global locale* affects various standard C++ functions that format or parse text, most directly the regular expression algorithms discussed later in this chapter and the I/O streams seen in Chapter 5. It is implementation dependent whether there is one program-wide global locale instance or one per thread of execution.

The global locale always starts out as the classic "C" locale. To set the global locale, you use the static `std::locale::global()` function. To get a copy of the currently active global locale, simply default-construct a `std::locale`. For example:

```
std::locale current_locale;
std::cout << '"' << current_locale.name() << '"' << '\n';  // "C"
std::cout << 100000 << '\n';  // 100000
std::locale::global(std::locale("")); // Global locale -> user preferences
std::cout << 100000 << '\n';  // 100000
std::cout.imbue(std::locale());        // Imbue the current global locale
std::cout << 100000 << '\n';  // Some possible outputs (locale dependent):
                              // 100,000; 100 000; 100.000; 1,00,000; ...
```

■ **Note** To avoid race conditions, standard C++ objects (such as newly created stream or `regex` objects) always copy the global `locale` upon construction. Calling `global()` therefore does not affect existing objects, including `std::cout` and the other standard streams of `<iostream>`. To change their locale, you must call their `imbue()` member.

Basic std::locale Members

The following table lists most basic functions offered by a `std::locale`, not including the copy members. More advanced members to combine or customize locales are discussed near the end of the section:

Member	Description
`global()`	Static function to set the active global locale (discussed earlier).
`classic()`	Static function returning a constant reference to a classic `"C"` `locale`.
`locale()`	Default constructor: creates a copy of the global locale.
`locale(name)`	Construction from locale name, as discussed earlier. Throws a `std::runtime_exception` if a nonexistent name is passed.
`name()`	Returns the locale name, if any. If the `locale` represents a customized or combined locale (discussed later), `"*"` is returned.
`==` / `!=`	Compares two `locale` objects. Customized or combined locales are equal only if they are the same object or one is a copy of the other.

Locale Facets

As obvious from the previous subsection, the `std::locale` public interface does not offer much functionality. All localization facilities are instead offered in the form of *facets*. Each `locale` object encapsulates a number of such facets, a reference to which may be obtained via the `std::use_facet<FacetType>()` function. The following example uses the classic locale's numeric punctuation facet to print out the locale's decimal mark for formatting floating-point numbers:

```
const auto& f =
   std::use_facet<std::numpunct<char>>(std::locale::classic());
std::cout << f.decimal_point() << std::endl;   // Prints a dot('.')
```

For all standard facets, the instance referred to by the result of `use_facet()` cannot be copied, moved, swapped, or deleted. This facet is (co-)owned by the given `locale` and is deleted together with the (last) `locale` that owns it. When requesting a `FacetType` the given `locale` does not own, a `bad_cast` exception is raised. To verify the presence of a facet, you can use `std::has_facet<FacetType>()`.

■ **Caution** Never do something like `auto& f = use_facet<...>(std::locale("..."));`: the facet `f` was owned by the temporary `locale` object, so using it will likely crash.

By default, `locales` contain specializations of all the facets introduced in the remainder of this section, each in turn specialized for at least the `char` and `wchar_t` character types (additional minimal requirements are discussed throughout the section).

Implementations may include more facets, and programs can even add custom facets themselves, as explained later.

We now discuss the 12 standard facet classes listed in Table 6-4 in order, grouped in sections by category. Afterward, we show how to combine facets of different locales and create customized facets. Although this is perhaps not something most programmers will use regularly, occasionally the need does arise to customize facets. Regardless, it is worth knowing the scope and various effects of localization and to keep them in mind when developing programs that show or process user text (i.e., most programs).

Table 6-4. *Overview of the 12 Basic Facet Classes, Grouped by Category*

Category	Facets
Numeric	`numpunct<C>`, `num_put<C>`, `num_get<C>`
Monetary	`moneypunct<C, International>`, `money_put<C>`, `money_get<C>`
Time	`time_put<C>`, `time_get<C>`
Ctype	`ctype<C>`, `codecvt<C1, C2, State>`
Collate	`collate<C>`
Messages	`messages<C>`

Numeric Formatting

The facets of the `numeric` and `monetary` categories follow the same pattern: there is one `punct` facet (short for *punctuation*) with the locale-specific formatting parameters, plus both a `put` and a `get` facet responsible for the actual formatting and parsing of values, respectively. The latter two facets are mostly intended to be used by the stream objects introduced in Chapter 5. The concrete format they use to read or write values is determined by a combination of the parameters set in the `punct` facet and others set using the stream's members or stream manipulators.

Numeric Punctuation

The `std::numpunct<CharT>` facet offers functions to retrieve the following information related to the formatting of numeric and Boolean values:

- `decimal_point()`: Returns the decimal separator
- `thousands_sep()`: Returns the thousands separator character
- `grouping()`: Returns a `std::string` encoding the *digit grouping*
- `truename()` and `falsename()`: Return `basic_string<CharT>`s with textual representations for Boolean values

In the lottery example at the beginning of the section, a numeric value of 100000.00 was formatted using three different locales: "`100,000.00`", "`1,00,000.00`", and "`100.000,00`". The first two locales use a comma (`,`) and dot (`.`) as thousands and decimal separator, respectively, whereas for the third it is the other way around.

The digit `grouping()` is encoded as a sequence of `char` values indicating the number of digits in each group, starting with the number in the rightmost group. The last `char` in the sequence is used for all subsequent groups as well. Most locales group digits in threes, for example, which is encoded as `"\3"`. (Note: Do not use `"3"`, because the `'3'` ASCII character results in a `char` with value 51; i.e., `'3' == '\51'`.) For Indian locales, however, as seen in "`1,00,000.00`", only the rightmost group contains three digits; all other groups contain only two. This is encoded as `"\3\2"`. To indicate an infinite group, a `std::numeric_limits<char>::max()` value may be used in the last position. An empty `grouping()` `string` denotes that no grouping should be used at all, which is the case, for instance, for the classic `"C"` locale.

Formatting and Parsing of Numeric Values

The `std::num_put` and `num_get` facets constitute the implementation of the `<<` and `>>` stream operators described in Chapter 5 and provide two sets of methods with the following signature:

```
Iter put(Iter target, ios_base& stream, char fill, X value)
Iter get(Iter begin, Iter end, ios_base& stream, iostate& error, X& result)
```

Here `X` can be `bool`, `long`, `long long`, `unsigned int`, `unsigned long`, `unsigned long long`, `double`, `long double`, or a `void` pointer. For `get()`, `unsigned short` and `float` are also possible. These methods either format a given numeric `value` or try to parse the characters in the range `[begin, end)`. In both cases, the `ios_base` parameter is a reference to a stream from which locale and formatting information is taken (including the stream's formatting flags and precision: see Chapter 5).

All `put()` functions simply return `target` after writing the formatted character sequence there. The `fill` character is used for padding if the formatted length is less than `stream.width()` (see Chapter 5 for the padding rules).

If parsing succeeds, `get()` stores the numeric value in `result`. If the input did not match the format, `result` is set to zero and the `failbit` is set in the `iostate` parameter (see Chapter 5). If the parsed value is too large/small for type `X`, the `failbit` is set as well, and `result` is set to `std::numeric_limits<X>::max()/lowest()` (see Chapter 1). If the end of the input was reached (can be a success or a failure), the `eofbit` is set. An iterator to the first character after the parsed sequence is returned.

We do not show example code here, but these facets are analogous to the monetary formatting facets introduced next, for which we do include a full example.

Monetary Formatting

Monetary Punctuation

The `std::moneypunct<CharType, International=false>` facet offers functions to retrieve the following information related to formatting monetary values:

- `decimal_point()`, `thousands_sep()`, and `grouping()`: Analogous to the numeric punctuation members seen earlier.
- `frac_digits()`: Returns the number of digits after the decimal separator. A typical value is 2.

- `curr_symbol()`: Returns the currency symbol, such as `'€'`, if the `International` template parameter is `false`, and the international currency code (usually three letters) followed by a space, such as `"EUR "`, if `International` is `true`.
- `pos_format()` and `neg_format()`: Return a `money_base::pattern` structure (discussed later) describing how positive and negative monetary values are to be formatted.
- `positive_sign()` and `negative_sign()`: Return a formatting `string` for positive and negative monetary values.

The latter four members need more explanation. They use types defined in `std::money_base`, a base class of `moneypunct`. The `money_base::pattern` structure, defined as `struct pattern{ char field[4]; }`, is an array containing four values of the `money_base::part` enumeration, with these supported values:

part	Description
`none`	Optional whitespace characters, except when `none` appears last.
`space`	At least one whitespace character.
`symbol`	The currency symbol, `curr_symbol()`.
`sign`	The first character returned by `positive_sign()` or `negative_sign()`. Additional characters appear at the end of the formatted monetary value.
`value`	The monetary value.

For example, assume that the `neg_format()` pattern is `{none, symbol, sign, value}`, that the currency symbol is `'$'`, that `negative_sign()` returns `"()"`, and that `frac_digits()` returns `2`. Then the value `-123456` is formatted as `"$(1,234.56)"`.

■ **Note** For American and many European locales, `frac_digits()` equals `2`, meaning unformatted values are to be expressed in, for example, cents rather than dollars or euros. This is not always the case, though: for the Japanese locale, for example, `frac_digits()` is `0`.

Formatting and Parsing of Monetary Values

The facets `std::money_put` and `money_get` handle formatting and parsing of monetary values and are mainly intended to be used by the `put_money()` and `get_money()` I/O manipulators discussed in Chapter 5. The facets offer methods of this form:

```
Iter put(Iter target, bool intl, ios_base& stream, char fill, X value)
Iter get(Iter begin, Iter end, bool intl, ios_base& stream,
                                          iostate& error, X& result)
```

Here X is either `std::string` or `long double`. The behavior and meaning of the parameters is similar to that discussed for `num_put` and `num_get` earlier. If `intl` is `false`, currency symbols like $ are used; otherwise, strings like "USD " are used.

The following illustrates how these facets can be used, although you normally simply use `std::put_/get_money()` (uses <cassert> and <sstream>):

```
std::locale my_locale("en-US");  // For Windows; for Linux use "en_US"
std::stringstream stream;
stream.imbue(my_locale);

// Perform equivalent of 'stream << std::put_money(valueIn);' explicitly:
long double valueIn = 123456;  // (Or: 'std::string valueIn = "123456";')
auto& money_formatter = std::use_facet<std::money_put<char>>(my_locale);
stream << std::showbase;
auto target = std::ostreambuf_iterator<char>(stream);
money_formatter.put(target, false, stream, ' ', valueIn);   // $1,234.56

// Perform equivalent of 'stream >> std::get_money(valueOut);' explicitly
long double valueOut;          // (Or: 'std::string valueOut;')
auto& money_parser = std::use_facet<std::money_get<char>>(my_locale);
std::ios_base::iostate error = std::ios_base::goodbit;
auto b = std::istreambuf_iterator<char>(stream);
auto e = std::istreambuf_iterator<char>();
money_parser.get(b, e, false, stream, error, valueOut);     // 123456
if (error != std::ios_base::goodbit) stream.setstate(error);

assert(valueIn == valueOut);
```

Time and Date Formatting

The two facets `std::time_get` and `time_put` handle parsing and formatting of time and dates and power the `get_time()` and `put_time()` manipulators seen in Chapter 5. They provide methods with the following signatures:

```
Iter put(Iter target, ios_base& stream, char fill, tm* value, <format>)
Iter get(Iter begin, Iter end, ios_base& stream, iostate& error,
         tm* result, <format>)
```

The <format> is either 'const char* from, const char* to', pointing to a time-formatting pattern expressed using the same syntax as explained for `strftime()` in Chapter 2, or 'char format, char modifier', a single time format specifier of the same grammar with optional modifier. The behavior and meaning of the parameters is analogous to those for the numeric and monetary formatting facets. The `std::tm` structure is explained in Chapter 2 as well. Only those members of the passed `tm` are used/written that are mentioned in the formatting pattern.

In addition to the generic get() functions, the time_get facet has a series of more restricted parsing functions, all with the following signature:

```
Iter get_x(Iter begin, Iter end, ios_base& stream, iostate& error, tm*)
```

Member	Description
get_time()	Tries to parse a time as %H:%M:%S.
get_date()	Tries to parse a date using a format that depends on the value of the facet's date_order() member: either no_order: %m%d%y, dmy: %d%m%y, mdy: %m%d%y, ymd: %y%m%d, or ydm: %y%d%m. This date_order() enumeration value reflects the locale's %X date format.
get_weekday() get_monthname()	Tries to parse a name for a weekday or month, possibly abbreviated.
get_year()	Tries to parse a year. Whether two-digit year numbers are supported depends on your implementation.

Character Classification, Transformation, and Conversion

Character Classification and Transformation

The ctype<CharType> facets offer a series of locale-dependent character classification and transformation functions, including equivalents for those of the <cctype> and <cwctype> headers seen earlier.

For use in the character classification functions listed next, 12 member constants of a bitmask type ctype_base::mask are defined (ctype_base is a base class of ctype), one for each character class. Their names equal the class names given in Table 6-1. Although their values are unspecified, alnum == alpha|digit and graph == alnum|punct. The following table lists all classification functions (input character ranges are represented using two CharType* pointers b and e):

Member	Description
is(mask,c)	Checks whether a given character c belongs to any of the character classes specified by mask.
is(b,e,mask*)	Identifies for each character in the range [b, e) the complete mask value that encodes all classes it belongs to, and stores the result in the output range pointed to by the last argument. Returns e.
scan_is(mask,b,e) scan_not(mask,b,e)	Scans the character range [b, e) and returns a pointer to the first character that belongs/does not belong to any of the classes specified by mask. If none is found, the result is e.

The same facets also offer these transformation functions:

Member	Description

`tolower(c)`
`toupper(c)`
`tolower(b,e)`
`toupper(b,e)`
	Performs upper-to-lower transformation or vice versa on a single character (result is returned) or a character range [`b`, `e`) (transformed in place; `e` is returned). Characters that cannot be transformed are left unchanged.

`widen(c)` `widen(b,e,o)`	Transforms `char` values to the facet's character type on a single character (result is returned) or a character range [`b`, `e`) (transformed characters are put in the output range starting at `*o`; `e` is returned). Transformed characters never belong to a class their source characters did not belong to.
`narrow(c,d)` `narrow(b,e,d,o)`	Transformation to `char`; opposite of `widen()`. However, only for the 96 *basic source characters* (all `space` and `printable` ASCII characters except `$`, `` ` ``, and `@`) the relation `widen(narrow(c,o)) == c` is guaranteed to hold. If no transformed character is readily available, the given default `char d` is used.

The `<locale>` header defines a series of convenience functions for those functions of the `ctype` facets that also exist in `<cctype>` and `<cwctype>`: `std::is`*class*`(c, locale&)`, with *class* a name from Table 6-1, and `tolower(c, locale&)` / `toupper(c, locale&)`. The implementations of the former set of functions, for instance, all have the following form:

```
template <typename CharT> bool isclass(CharT c, const std::locale& l) {
   return std::use_facet<std::ctype<CharT>>(l).is(class,c);
}
```

Character-Encoding Conversions

A `std::codecvt` facet converts character sequences between two *character encodings*. This is explained earlier in "Character-Encoding Conversion," because these facets are useful also outside the context of locales. Each `std::locale` contains at least instances of the four `codecvt` specializations listed in Table 6-2, which implement potentially locale-specific converters. These are used implicitly by the streams of Chapter 5 when converting, for example, between wide and narrow strings. Because directly using these low-level facets is not recommended, we do not explain their members here. Always use the helper classes discussed in the "Character-Encoding Conversion" section instead.

String Ordering and Hashing

The `std::collate<CharType>` facet implements the following locale-dependent string-ordering comparisons and hashing functions. All character sequences are specified using begin (inclusive) and end (exclusive) `CharType*` pointers:

Member	Description
`compare()`	Locale-dependent three-way comparison of two character sequences, returning `-1` if the first precedes the second, `0` if both are equivalent, and `+1` otherwise. Not necessarily the same as naïve lexicographical sequence comparison.
`transform()`	Transforms a given character sequence to a specific normalized form, which is returned as a `basic_string<CharType>`. Applying naïve lexicographical ordering on two transformed strings (as with their `operator<`) returns the same result as applying the facet's `compare()` function on the untransformed sequences.
`hash()`	Returns a `long` hash value for the given sequence (see Chapter 3 for hashing) that is the same for all sequences that `transform()` to the same normalized form.

A `std::locale` itself is also a `std::less<std::basic_string<CharT>>`-like functor (see Chapter 2) that compares two `basic_string<CharT>`s using its `collate<CharT>` facet's `compare()` function. The following example sorts French strings first using `operator<` of `std::string`, and then using a French locale (the locale name to use is platform specific). In addition to `<locale>`, this code requires `<vector>`, `<string>`, and `<algorithm>`:

```
std::vector<std::string> strings = { "liberté", "égalité", "fraternité" };
auto printSortedMotto = [&strings] {
   for (auto& s : strings) std::cout << s << ' ';
   std::cout << std::endl;
};
std::sort(begin(strings), end(strings));       // Lexicographic sort
printSortedMotto();                    // fraternité liberté égalité
std::sort(begin(strings), end(strings), std::locale("french"));
printSortedMotto();                    // égalité fraternité liberté
```

Message Retrieval

The `std::messages<CharT>` facet facilitates retrieval of textual messages from *message catalogs*. These catalogs are essentially associative arrays that map a series of integers to a localized string. This could in principle be used, for instance, to retrieve translated error messages based on, for example, their error category and code (see Chapter 8). Which catalogs are available, and how they are structured, is entirely platform specific. For some, standardized message catalog APIs are used (such as POSIX's `catgets()` or

GNU's gettext()), whereas others may not offer any catalogs (this is typically the case for Windows). The facet offers these functions:

Member	Description
open(n,l)	Opens a catalog based on a given platform-specific string n (a basic_string<CharT>), and for the given std::locale l. Returns a unique identifier of some signed integer type catalog.
get(c,set,id,def)	Retrieves from the catalog with given catalog identifier c, the message identified by set and id (two int values whose interpretation is catalog specific), and returns it as a basic_string<CharT>. Returns def if no such message is found.
close(c)	Closes the catalog with the given catalog identifier c.

Combining and Customizing Locales

The constructs of the <locale> library are designed to be very flexible when it comes to combining or customizing locale facets.

Combining Facets

std::locale provides combine<FacetType>(const locale& c), which returns a copy of the locale on which combine() is called, except for the FacetType facet, which is copied from the given argument. Here is an example (the put_money() I/O manipulator requires the <iomanip> header):

```
using namespace std;

int bigValue = 10000;
long double money = 123456;
cout << bigValue << " " << put_money(money) << '\n'; // 10000 123456

locale chinese("zh_CN");  // For Windows use "zh-CN"
cout.imbue(chinese);
cout << bigValue << ' ' << put_money(money) << '\n'; // 10,000 1,234.56

// Use the neutral "C" locale, but with Chinese monetary punctuation:
locale combined = locale::classic().combine<moneypunct<char>>(chinese);
cout.imbue(combined);
cout << bigValue << ' ' << put_money(money) << '\n'; // 10000  1,234.56
```

Alternatively, `std::locale` has a constructor accepting a base `locale` and an overriding facet that does the same as `combine()`. For example, the creation of `combined` in the previous example can be expressed as follows:

```
locale combined(locale::classic(), &use_facet<moneypunct<char>>(chinese));
```

`std::locale` moreover has a number of constructors to override all facets of one or more categories at once (*`String`* is either a `std::string` or a C-style string representing the name of a specific locale):

```
locale(const locale& base, String name, category cat)
locale(const locale& base, const locale& overrides, category cat)
```

For each of the six categories listed in Table 6-4, `std::locale` defines a constant with that name. The `std::locale::category` type is a bitmask type, meaning categories can be combined using bitwise operators. The `all` constant, for example, is defined as `collate | ctype | monetary | numeric | time | messages`. These constructors can be used to create a `combined` facet similar to the one earlier:

```
locale combined(locale::classic(), chinese, locale::monetary);
```

Custom Facets

All public functions *`func`*`()` of the facets simply call a protected virtual method on the facet called `do_`*`func`*`()`.[3] You can implement custom facets by inheriting from existing ones and overriding these `do`-methods.

This first simple example changes the behavior of the `numpunct` facet to use the strings `"yes"` and `"no"` instead of `"true"` and `"false"` for Boolean input and output:

```
class yes_no_numpunct : public std::numpunct<char> {
protected:
   virtual string_type do_truename() const override { return "yes"; }
   virtual string_type do_falsename() const override { return "no"; }
};
```

You can use this custom facet, for instance, by imbuing it on a stream. The following prints `"yes / no"` to the console:

```
std::cout.imbue(std::locale(std::cout.getloc(), new yes_no_numpunct));
std::cout << std::boolalpha << true << " / " << false << std::endl;
```

[3] *Nearly* all functions: for performance, `is()`, `scan_is()`, and `scan_not()` of the `ctype<char>` specialization do not call a virtual function, but perform lookups in a `mask*` array (`ctype::classic_table()` for the `"C"` locale). A custom instance may be created by passing a custom lookup array to the facet's constructor.

Recall that facets are reference counted and that the destructor of the `std::locale` hence properly cleans up your custom facet.

The disadvantage of deriving from facets such as `numpunct` and `moneypunct` is that those generic base classes implement locale-independent behavior. To start from a locale-specific facet instead, facet classes such as `numpunct_byname` are available. For all facets seen so far, except the `numeric` and `monetary` `put` and `get` facets, a facet subclass exists with the same name but appended with `_byname`. They are constructed passing a locale name (`const char*` or `std::string`) and then behave as if taken from the corresponding `locale`. You can override from these facets to modify only specific aspects of a facet for a given locale.

The next example modifies the monetary punctuation facet to facilitate output using a format standard in accounting: negative numbers are put between parentheses, and padding is done in a particular way. You do so without overriding a locale's currency symbol or most other settings by starting from `std::moneypunct_byname` (`string_type` is defined in `std::moneypunct`):

```
class accounting_moneypunct : public std::moneypunct_byname<char, false> {
public:
   accounting_moneypunct(const std::string& name)
      : moneypunct_byname(name) { }
protected:
   // Put negative numbers between parentheses:
   virtual string_type do_negative_sign() const override { return "()"; }
   // Override formats to facilitate accounting-style padding:
   static pattern acc_format() { return { symbol, space, sign, value }; }
   virtual pattern do_neg_format() const override { return acc_format(); }
   virtual pattern do_pos_format() const override { return acc_format(); }
};
```

This facet may then be used as follows (see Chapter 5 for details on the stream I/O manipulators of `<iomanip>`):

```
const auto name = "en_US";    // Use platform specific locale name...
std::locale my_locale(std::locale(name), new accounting_moneypunct(name));
std::cout.imbue(my_locale);
std::cout << std::showbase << std::internal; //show $ sign + tweak padding
for (auto val : { 100000, -500 })
   std::cout << std::setw(12) << std::put_money(val) << '\n';
```

The output of this program should be

```
$   1,000.00
$     (5.00)
```

You can in theory create a new facet class by directly inheriting from `std::facet` and add it to a `locale` using the same constructor to use it in your own library code later. The only additional requirement is that you define a default-constructed static constant named `id` of type `std::locale::id`.

C Locales **<clocale>**

Locale-sensitive functions from the C Standard Library (including most functions in <cctype> and the I/O operations of <cstdio> and <ctime>) are not directly affected by the global C++ `locale`. Instead, they are governed by a corresponding C locale. This C locale is changed by one of two functions:

- `std::locale::global()` is guaranteed to modify the C locale to match the given C++ `locale`, as long as the latter has a name. Otherwise, its effect on the C locale, if any, is implementation-defined.
- Using the `std::setlocale()` function of <clocale>. This does not affect the C++ global `locale` in any way.

In other words, when using standard locales, a C++ program should simply call `std::locale::global()`. To write portable code when combining multiple locales, however, you have to call both the C++ and the C function because not all implementations set the C locale as expected when changing the `global()` C++ locale to a combined `locale`. This is done as follows:

```
// Use the user's preferred locale settings,
// but with neutral numeric and monetary formatting
std::locale::global(std::locale(std::locale(""), "C",
                                std::locale::numeric | std::locale::monetary));
std::setlocale(LC_ALL, "");
std::setlocale(LC_NUMERIC, "C");
std::setlocale(LC_MONETARY, "C");
```

The `setlocale()` function takes a single category number (not a bitmask type; supported values include at least `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_MONETARY`, `LC_NUMERIC`, and `LC_TIME`) and a locale name, all analogous to their C++ equivalents. It returns the name of the active C locale upon success as a `char*` pointer into a reused, global buffer, or `nullptr` upon failure. If `nullptr` is passed for the locale name, the C locale is not modified.

Unfortunately, the C locale functionality is far less powerful than the C++ one: customized facets or selecting individual facets for combining is not possible, making the use of such advanced locales impossible with portable code in general.

The <clocale> header has one more function: `std::localeconv()`. It returns a pointer to a global `std::lconv` `struct` with public members equivalent to the functions of the `std::numpunct` (`decimal_point`, `thousands_sep`, `grouping`) and `std::moneypunct` facets (`mon_decimal_point`, `mon_thousands_sep`, `mon_grouping`, `positive_sign`, `negative_sign`, `currency_symbol`, `frac_digits`, etc.). These values should be treated as read-only: writing to them results in undefined behavior.

Regular Expressions <regex>

A *regular expression* is a textual representation of a pattern or patterns to be matched against a *target sequence* of characters. The regular expression `ab*a`, for instance, matches any target sequence starting with the character `a`, followed by zero or more `b`s, and ending again with an `a`. Regular expressions can be used to search for or replace particular patterns in the target, or to verify that it matches a desired pattern. You see how to perform these operations using the `<regex>` library later; first we introduce how to form and create regular expressions.

The ECMAScript Regular Expression Grammar

The syntax used to express patterns in textual form is defined by a *grammar*. By default, `<regex>` uses a modified version of the grammar used by the ECMAScript scripting language (best known for its widely used dialects JavaScript, JScript, and ActionScript). What follows is a concise, comprehensive reference for this grammar.

A regular expression *pattern* is a *disjunction* of sequences of *terms*, with each term either an *atom*, an *assertion*, or a *quantified atom*. Supported atoms and assertions are listed in Tables 6-5 and 6-6, and Table 6-7 shows how atoms are quantified to express repetitive patterns. These terms are concatenated without separators and then optionally combined into disjunctions using the `|` operator. Empty disjuncts are allowed, with *pattern*`|` matching either the given pattern or the empty sequence. Some examples should clarify:

- `\r\n?|\n` matches line-break sequences for all major platforms (i.e., `\r`, `\r\n`, or `\n`).
- `<(.+)>(.*)</\1>` matches XML-like sequences of the form *<TAG>anything</TAG>* using a back reference for matching the closing tag, and extra grouping in the middle to allow retrieval of the second submatch (discussed later).
- `(?:\d{1,3}\.){3}\d{1,3}` matches IPv4 addresses. This naïve version also matches illegal addresses, though, such as `999.0.0.1`, and the poor grouping prohibits the four matched numbers from being retrieved afterward. Note that without the `?:`, `\1` still would only refer to the third matched number.

■ **Tip** When entering regular expressions as string literals in a C++ program, all backslashes have to be escaped. The first example becomes `"\\r\\n?|\\n"`. Because this is both tedious and obscuring, we recommend using raw string literals instead: for instance, `R"(\r\n?|\n)"`. Remember that the surrounding parentheses are part of the raw string literal notation and do not constitute a regular expression group.

The difference between an atom and an assertion is that the former consumes characters from the target sequence (typically one), whereas the latter does not. The (quantified) atoms in a pattern consume target characters one by one, simultaneously progressing left to right through both the pattern and target sequences. For an assertion to match, a specific condition must hold on the current position in the target (think of it as the caret position when typing text).

Character Classes

A *character class* is a `[`*d*`]` or `[^`*d*`]` atom that defines a set of characters it may (for `[`*d*`]`) or may not (`[^`*d*`]`) match. The class definition *d* is a sequence of *class atoms*, each one either

- An individual character.
- A character range of the form *from*`-`*to* (bounds are inclusive).
- Starting with a backslash (`\`): the equivalent of any atom from Table 6-5 except back references, with the obvious meaning. Note that characters such as `* + . $` do not need escaping in this context, but characters `- [ ] : ^` may. Also, inside class definitions, `\b` denotes the backspace character (`\u0008`).
- One of three types of special character class atoms enclosed between nested square brackets (described shortly).

The descriptors are concatenated without separators. For example, `[_a-zA-Z]` matches either an underscore or a single character in the range a–z or A–Z, whereas `[^\d]` matches any single character that is not a decimal digit.

The first special class atom has form `[:`*name*`:]`. At least the following names are supported: equivalents of all 12 character classes explained in the section on character classification—`alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, and `xdigit`—and `d`, `s`, and `w`. Of the latter, `d` and `s` are short for `digit` and `space`, and `w` is the class of *word characters* with `[:w:]` equivalent to `_[:alnum:]` (mind the underscore!). That is, for the classic `"C"` locale, `[[:w:]] == [_a-zA-Z]`. As another example, `[\D] == [^\d] == [^[:d:]] == [^[:digit:]] == [^0-9]`.

The second type of special class atoms looks like `[.`*name*`.]`, where *name* is a locale- and implementation-specific *collating element* name. This name can be a single character *c*, in which case `[[.`*c*`.]]` is equivalent to `[`*c*`]`. Similarly, `[[.comma.]]` may equal `[,]`. Some names refer to multicharacter collating elements, that is, multiple characters that are considered a single character in a specific alphabet and its sorting order. Possible names for the latter include those of digraphs: `ae`, `ch`, `dz`, `ll`, `lj`, `nj`, `ss`, and so on. For instance, `[[.ae.]]` matches two characters, whereas `[ae]` matches one.

Class atoms of the form `[=`*name*`=]`, finally, are similar to `[.`*name*`.]`, except that they match all characters that are part of the same *primary equivalence class* as the named collating element. Essentially, this means `[=e=]` in French should match not only *e* but also *é*, *è*, *ê*, *E*, *É*, and so on. Similarly, `[=ss=]` in German should match the digraph *ss*, but also the Eszett character (ß).

Greedy vs. Non-greedy Quantification

By default, quantified atoms as defined in Table 6-7 are *greedy*: they first match sequences that are as long as possible and only try shorter sequences if that does not lead to a successful match. To make them *non-greedy*—that is, to make them try the shortest possible sequences first—add a question mark (`?`) after the quantifier.

Recall, for example, the earlier example `"<(.+)>(.*)</\1>"`. When searching for or replacing its first match in `"<b>Bold</b>, not bold, <b>bold again</b>"`, this pattern matches the full sequence. The non-greedy version, `"<(.+)>(.*?)</\1>"`, instead matches only the desired `"<b>Bold</b>"`.

As an alternative to a non-greedy quantifier, a negative character class may be considered as well (it may be more efficient), such as `"<(.+)>([^<]*)</\1>"`.

Regular Expression Objects

The `<regex>` library models regular expressions as `std::basic_regex<CharT>` objects. Of this, at least two specializations are available for use with narrow strings (`char` sequences) and wide strings (`wchar_t` sequences): `std::regex` and `std::wregex`. The examples use `regex`, but `wregex` is completely analogous.

Construction and Syntax Options

A default-constructed `regex` does not match any sequence. More useful regular expressions are created using the constructors of the following form:

```
regex(Pattern, regex::flag_type flags = regex::ECMAScript);
```

The desired regular expression `Pattern` may be represented as either a `std::string`, a null-terminated `char*` array, a `char*` buffer with a `size_t` length (the number of `char`s to be read from the buffer), an `initializer_list<char>`, or a range formed by a beginning and end iterator.

When the given pattern is invalid (mismatched parentheses, a bad back reference, etc.), a `std::regex_error` is thrown. This is a `std::runtime_exception` with an additional `code()` member returning one of 11 error codes of type `std::regex_constants::error_type` (`error_paren`, `error_backref`, etc.).

The last argument determines which grammar is used and may be used to toggle certain syntax options. The `flag_type` values are aliases for those of `std::regex_constants::syntax_option_type`. Because it is a bitmask type, its values may be combined using the `|` operator. The following syntax options are supported:

Option	Effect
`collate`	Character ranges of form `[a-z]` become locale sensitive. For a French locale, for instance, `[a-z]` should then match *é*, *è*, etc.
`icase`	Character matches are done in a case-insensitive manner.
`multiline` C++17	The ^ and $ assertions (Table 6-6) are guaranteed to match the beginning and end of a line, respectively. This option is only valid for the `ECMAScript` grammar.
`nosubs`	No submatches against subexpressions are stored in `match_results` (discussed later). Back references will likely fail as well.
`optimize`	Hints the implementation to prefer improved matching speed over performance during construction of regular expression objects.
`ECMAScript`	Uses the ECMAScript-based regular expression grammar (default).
`basic`	Uses the POSIX basic regular expression grammar (BRE).
`extended`	Uses the POSIX extended regular expression grammar (ERE).
`grep`	Uses the grammar of the POSIX utility `grep` (a BRE variant).
`egrep`	Uses the grammar of the POSIX utility `grep -E` (an ERE variant).
`awk`	Uses the grammar of the POSIX utility `awk` (another ERE variant).

Of the last six options, only one is allowed to be specified; if none is specified, `ECMAScript` is used by default. All POSIX grammars are older and less powerful than the ECMAScript grammar. The only reason to use them would therefore be that you are already familiar with them or have preexisting regular expressions. Either way, there is no reason to detail these grammars here.

Basic Member Functions

A `regex` object is primarily intended to be passed to one of the global functions or iterator adaptors explained later, so not many member functions operate on it:

- A `regex` can be copied, moved, and swapped.
- It can be (re)initialized with a new regular expression and optional syntax options using `assign()`, which has the exact same set of signatures as its nondefault constructors.
- The `flags()` member returns the syntax options flag it was initialized with, and `mark_count()` returns the number of marked subexpressions in its regular expression (see Table 6-5).
- The regex `std::locale` is returned by `getloc()`. This affects matching behavior in several ways and is initialized with the active global C++ locale upon construction. After construction, it may be changed using the `imbue()` function.

Matching and Searching Patterns

The `std::regex_match()` function verifies that the *full target sequence* matches a given pattern, whereas the similar `std::regex_search()` searches for a *first occurrence* of a pattern in the target. Both return `false` if no successful match is found. These function templates have an analogous set of overloads, all with signatures of this form:

```
bool regex_match (Target [, Results&], const Regex&, match_flag_type = 0);
bool regex_search(Target [, Results&], const Regex&, match_flag_type = 0);
```

All but the last argument is templated on the same character type `CharT`, with implementations available for at least `char` and `wchar_t`. As for the arguments

- A typical combination for the first three arguments is `(w)string`, `(w)smatch`, `(w)regex`.
- Instead of a `basic_string<CharT>`, the `Target` sequence may also be represented as a null-terminated `CharT*` array (used also for string literals) or a pair of bidirectional iterators that mark the bounds of a `CharT` sequence. In both these cases, the normal `Results` type becomes `std::(w)cmatch`.
- The `w?[sc]match` types used for the optional match `Results` output argument are discussed in the next subsection.
- The `Regex` object passed is not copied, so these functions must not (ideally cannot) be called using a temporary object.
- To control matching behavior, a value of the bitmask type `std::regex_constants::match_flag_type` may be passed. Supported values are shown in the following table:

Match Flag	Effect
`match_default`	Default matching behavior is used (this constant has value zero).

`match_not_bol`
`match_not_eol`
`match_not_bow`
`match_not_eow`
	The first or last position in the target sequence is no longer considered the **b**eginning/**e**nd **o**f a **l**ine/**w**ord. Affects the ^, $, \b, and \B annotations as explained in Table 6-6.

`match_any`	If multiple disjuncts of a disjunction match, it is not required to find the longest match among them: any match will do (e.g., the first one found, if that speeds things up). Not relevant for the ECMAScript grammar, because this already prescribes the use of the leftmost successful match for disjunctions.
`match_not_null`	The pattern will not match the empty sequence.
`match_continuous`	The pattern only matches sequences that start at the beginning of the target sequence (implied for `regex_match()`).

(*continued*)

Match Flag	Effect
`match_prev_avail`	When deciding on line and word boundaries for `^`, `$`, `\b`, and `\B` annotations, matching algorithms look at the character at `--first`, with `first` pointing to the start of the target sequence. When set, `match_not_bol` and `match_not_bow` are ignored. Useful when repeatedly calling `regex_search()` on consecutive target subsequences. The iterators explained later do this correctly and are the recommended way to enumerate matches.

If either algorithm fails, a `std::regex_error` is raised. Because the regular expression's syntax is already verified upon construction of the `regex` object (see earlier), this only rarely occurs for very complex expressions if the algorithm runs out of resources.

Match Results

A `std::match_results<CharIter>` is effectively a sequential container (see Chapter 3) of `sub_match<CharIter>` elements, which are `std::pairs` of bidirectional `CharIters` pointing into the target sequence marking the bounds of the submatch sequences. At index 0, there is a `sub_match` for the full match, followed by one `sub_match` per marked subexpression in the order their opening parentheses appear in the regular expression (see Table 6-5). The following template specializations are provided:

Target	match_results	sub_match	CharIter
`std::string`	`std::smatch`	`std::ssub_match`	`std::string::const_iterator`
`std::wstring`	`std::wsmatch`	`std::wssub_match`	`std::wstring::const_iterator`
`const char*`	`std::cmatch`	`std::csub_match`	`const char*`
`const wchar_t*`	`std::wcmatch`	`std::wcsub_match`	`const wchar_t*`

std::sub_match

In addition to the `first` and `second` members inherited from `std::pair`, `sub_matches` have a third member variable called `matched`. This Boolean is `false` if the match failed or if the corresponding subexpression did not participate in the match. The latter occurs, for example, if the subexpression was part of a nonmatched disjunct, or of a nonmatched atom quantified with, for example, `?`, `*`, or {`0,`*n*}. When matching `"(a)?b|(c)"` against `"b"`, for instance, the match succeeds with a `match_result` that contains two empty `sub_matches` with `matched == false`.

The operations available for sub_matches are summarized in this table:

Operation	Description
length()	The length of the match sequence (0 if not matched).
str() / cast operator	Returns the match sequence as a std::basic_string.
compare()	Returns 0 if the sub_match compares equal to, and a positive/negative number if it compares greater/smaller than, a given sub_match, basic_string or null-terminated character array.
==, !=, <, <=, >, >=	Non-member operators for compare()ing between a sub_match and a sub_match, basic_string or character array, or vice versa.
<<	Non-member operator for streaming to an output stream.

std::match_results

A match_results can be copied, moved, swapped, and compared for equality using == and !=. In addition to those operations, the following member functions are available (functions related to custom allocators are omitted). Note that, unlike for strings, size() and length() are not equivalent here:

Operation	Description
ready()	A default-constructed match_results is not ready and becomes ready after execution of a match algorithm.
empty()	Returns size() == 0 (true if not ready() or after a failed match).
size()	Returns the number of sub_matches contained (one plus the number of marked subexpressions) if ready() and the match was successful, or zero otherwise.
max_size()	The theoretical maximum size() due to implementation or memory limitations.
operator[]	Returns the sub_match with specified index *n* (see earlier) or an empty sub_match sub with sub.matched == false if *n* >= size().
length(size_t=0)	results.length(*n*) is equivalent to results[*n*].length().
str(size_t=0)	results.str(*n*) is equivalent to results[*n*].str().
position(size_t=0)	The distance between the start of the target sequence and results[*n*].first.
prefix()	Returns a sub_match ranging from the start of the target sequence (inclusive) until that of the match (noninclusive). Always empty for regex_match(). Undefined if not ready().

(*continued*)

Operation	Description
`suffix()`	Returns a `sub_match` ranging from the end of the full match (noninclusive) until the end of the target sequence (inclusive). Always empty for `regex_match()`. Undefined if not `ready()`.
`begin()`, `cbegin()`, `end()`, `cend()`	Return iterators pointing to the first or one past the last `sub_match` contained in the `match_results`.
`format()`	Formats the matched sequence according to a specified format. The different overloads (either string- or iterator-based) have output, pattern, and format flag arguments analogous to those of the `std::regex_replace()` function explained later. Any `match_xxx` flags are ignored; only `format_yyy` flags are used.

Example

The following example illustrates the use of `regex_match()`, `regex_search()`, and `match_results` (`smatch`):

```
std::regex pattern(R"(<(.+)>(.*?)</\1>)");
std::string target = "<b>Bold</b>, not bold, <b>bold again</b>.";

std::cout << std::boolalpha;          // print true/false instead of 1/0
std::cout << std::regex_match(target, pattern) << "\n\n";       // false

std::smatch results;
auto begin = target.cbegin(), end = target.cend();
while (std::regex_search(begin, end, results, pattern)) {
   std::cout << results.str(2) << '\n';     // "Bold", then "bold again"
   begin += results.length();
}
```

But the preferred way of enumerating all matches is to use the iterators discussed in the next subsection.

Match Iterators

The `std::regex_iterator` and `regex_token_iterator` classes facilitate traversing all matches of a pattern in a target sequence. Like `match_results`, both are templated with a type of character iterator (`CharIter`). Four analogous type aliases also exist for the most

common cases: the iterator type prefixed with `s`, `ws`, `c`, or `wc`. The `while` loop from the example at the end of the previous subsection, for instance, may be rewritten as follows:

```
std::sregex_iterator begin(target.cbegin(), target.cend(), pattern),
                     end;   // default constructor creates end-iterator
std::for_each(begin, end, [](auto& results)    /* const std::smatch& */
                          { std::cout << results.str(2) << '\n'; });
```

In other words, a `regex_iterator` is a forward iterator that enumerates all `sub_matches` of a pattern as if found by repeatedly calling `regex_search()`. The previous `for_each()` loop is not only shorter and clearer though, it is also more correct in general than our naïve `while` loop: the iterator, for one, sets the `match_prev_avail` flag after the first iteration. Only one nontrivial constructor is available, creating a `regex_iterator<CharIter>` pointing to the first `sub_match` (if any) of a given `Regex` in the target sequence bounded by two bidirectional `CharIters`:

```
regex_iterator(CharIter, CharIter, const Regex&, match_flag_type = 0);
```

Analogous to a `regex_iterator`, which enumerates `match_results`, a `regex_token_iterator` enumerates all or specific `sub_matches` contained in these `match_results`. The same example, for instance, may be written as

```
std::sregex_token_iterator beg(target.cbegin(), target.cend(), pattern, 2),
                           end;  // default construction --> end-iterator
std::for_each(beg, end, [](auto& subMatch)   /* const std::ssub_match& */
                        { std::cout << subMatch << '\n'; });
```

The constructors of `regex_token_iterator` are analogous to the constructor of `regex_iterator` but have an extra argument indicating which `sub_matches` to enumerate. Overloads are defined for a single `int` (as in the example), `vector<int>`, `int[`*N*`]`, and `initializer_list<int>`. Replacing the `2` in the example with `{0,1}`, for example, outputs `"<b>Bold</b>"`, `"b"`, `"<b>bold again</b>"`, and then `"b"`. When omitted, this argument defaults to `0`, indicating only full pattern `sub_matches` are to be enumerated (the example then prints `"<b>Bold</b>"` and `"<b>bold again</b>"`).

Tokenizing

The last parameter of a `regex_token_iterator` can also be `-1` which turns it into a *field splitter* or *tokenizer*. This is a safe alternative to the C function `strtok()` from `<cstring>`. In this mode, a `regex_token_iterator` iterates over all subsequences that do *not* match the regular expression pattern. It can, for instance, be used to split a comma-separated string. In the following example, we allow comma separators to be preceded or followed by whitespace as well:

```
const std::string csv = "a, b, c,123";
const std::regex regex(R"(\s*,\s*)");
```

```
std::sregex_token_iterator beg(begin(csv), end(csv), regex, -1), end;
std::for_each(beg, end, [](auto& token) { std::cout << token << '\n'; });
```

Replacing Patterns

The final regular expression algorithm, `std::regex_replace()`, replaces all matches of a given pattern with another. The signatures are as follows:

```
String regex_replace(Target, Regex&, Format, match_flag_type = 0);
Out regex_replace(Out, Begin, End, Regex&, Format, match_flag_type = 0);
```

As before, argument types are templated in the same character type `CharT`, with support for at least `char` and `wchar_t`. The replacement `Format` is represented as either a `(w)string` or a null-terminated C-style string. For the target sequence, there are two groups of overloads. Those in the first represent the `Target` as a `(w)string` or a C-style string and return the result as a `(w)string`. Those in the second denote the target using bidirectional `Begin` and `End` character iterators and copy the result into an output iterator `Out`. The return value for the latter is an iterator pointing to one past the last character that was outputted.

All matches of the given `Regex` are replaced with the `Format` sequence, which by default may contain the following special character sequences:

Format	Replacement
$*n*	A copy of the *n*th marked subexpression of the match, where $n > 0$ is counted as with back references: see Table 6-5.
$&	A copy of the entire match.
$`	A copy of the prefix, the part of the target that precedes the match.
$´	A copy of the suffix, the part of the target that follows the match.
$$	A $ character (this is the only escaping required).

Analogously to earlier, only if the algorithm has insufficient resources to evaluate the match, a `std::regex_error` is thrown.

The following code, for example, prints `"d*v*w*l*d"` and `"debolded"`:

```
std::regex vowels("[aeiou]");
std::cout << std::regex_replace("devoweled", vowels, "*") << '\n';

std::regex bolds("<b>(.*?)</b>");
std::string target = "<b>debolded</b>";
std::ostream_iterator<char> out(std::cout);
std::regex_replace(out, target.cbegin(), target.cend(), bolds, "$1");
```

The final argument is again a `std::regex_constants::match_flag_type`, which for `regex_replace()` can be used to tweak both the matching behavior of the regular expression—using the same `match_xxx` values as listed earlier—and the formatting of the replacement. For the latter, the following values are supported:

Format Flag	Effect
`format_default`	Default formatting is used (this constant has value zero).
`format_sed`	The same syntax as the POSIX utility `sed` is used for the `Format`.
`format_no_copy`	Parts of the `Target` sequence that are not matches of the regular expression pattern are not copied to the output.
`format_first_only`	Only the first occurrence of the pattern is replaced.

Most of the atoms in Table 6-5 match a single character; only subexpressions and back references may match a sequence. Any other single character is also an atom that matches simply that character. All atoms may be quantified as shown in Table 6-7.

Table 6-5. *All Atoms with a Special Meaning in the ECMAScript Grammar*

Atom	Matches
`.`	Any single character except line terminators.[4]
`\0`, `\f`, `\n`, `\r`, `\t`, `\v`	One of the common control characters: null, form feed (FF), line feed (LF), carriage return (CR), horizontal tab (HT), and vertical tab (VT).
`\c`*letter*	The control character whose code unit equals that of the given ASCII lowercase or uppercase *letter* modulo 32. E.g., `\cj` == `\cJ` == `\n` (LF) as (code unit of `j` or `J`) % 32 = (106 or 74) % 32 = 10 = code unit of LF.
`\x`*hh*	The ASCII character with hexadecimal code unit *hh* (exactly two hexadecimal digits). E.g., `\x0A` == `\n` (LF), and `\x6A` == `J`.
`\u`*hhhh*	The Unicode character with hexadecimal code unit *hhhh* (exactly four hexadecimal digits). E.g., `\u006A` == `J`, and `\u03c0` == π (Greek letter pi).
[*class*]	A character of a given *class* (see main text): `[abc]`, `[a-z]`, `[[:alpha:]]`, etc.
[^*class*]	A character not of a given *class* (see main text). E.g., `[^0-9]`, `[^[:s:]]`, etc.
`\d`	A decimal digit character (short for `[[:d:]]` or `[[:digit:]]`).
`\s`	A whitespace character (short for `[[:s:]]` or `[[:space:]]`).
`\w`	A word character, i.e., an alphanumeric or underscore character (short for `[[:w:]]` or `[_[:alnum:]]`).

(*continued*)

Table 6-5. (*continued*)

Atom	Matches
`\D`, `\S`, `\W`	Complement of `\d`, `\s`, `\w`. In other words, any character that is not a decimal digit, whitespace, or word character, respectively (short for `[^[:d:]]`, etc.).
`\`*character*	The given *character*, as is. Required only for `\ . * + ? ^ $ ( ) [ ] { } |` because without escaping, these have special meaning; but any *character* may be used as long as `\`*character* has no special meaning.
(*pattern*)	Matches *pattern* and creates a *marked subexpression*, turning it into an atom that can be quantified, for one. The sequence it matches (called a *submatch*) can be retrieved from a `match_results` or referred to using a back reference (discussed later), either further in the surrounding pattern or in the replacement pattern when using `regex_replace()`.
(`?:`*pattern*)	Same as previous, but the subexpression is *unmarked*, meaning the submatch is not stored in a `match_results`, nor can it be referred to.
`\`*integer*	A *back reference*: matches the exact same sequence as the marked subexpression with index *integer* did earlier. Subexpressions are counted left to right in the order their opening parentheses appear in the full pattern, starting from one (recall: `\0` matches the null character).

Assertions, listed in Table 6-6, do not consume any characters, but simply add conditions for a successful pattern match. The `match_xxx` flags are optionally passed to the matching functions or iterators.

Table 6-6. *Assertions Supported by the ECMAScript Grammar*

Assertion	Matches If the Current Position Is …
`^`	The beginning of the target (unless `match_not_bol` is specified), or a position that immediately follows a line-terminator character[4] (the latter is only guaranteed to work if `multiline` is specified).
`$`	The end of the target (unless `match_not_eol` is specified), or the position of a line-terminator character (the latter is only guaranteed to work if `multiline` is specified).
`\b`	A word boundary: the next character is a word character,[5] whereas the previous is not, or vice versa. The beginning and end of the target are also word boundaries if the target begins/ends with a word character (and `match_not_bow`/`match_not_eow` is not specified, respectively).

(*continued*)

[4] A line terminator is one of four characters: line feed (`\n`), carriage return (`\r`), line separator (`\u2028`), or paragraph separator (`\u2029`).

[5] A word character is any character in the `[[:w:]]` or `[_[:alnum:]]` class: i.e., an underscore or any alphabetic or numerical digit character.

Table 6-6. (*continued*)

Assertion	Matches If the Current Position Is …
`\B`	Not a word boundary: both the previous and next characters are either word or nonword characters. See `\b` for when the beginning and end of the target are word boundaries.
(`?=`*pattern*)	A position at which the given *pattern* could be matched next. This is called a *positive lookahead*.
(`?!`*pattern*)	A position at which the given *pattern* would not be matched next. This is called a *negative lookahead*.

To make the quantifiers in Table 6-7 non-greedy, add an extra `?` after the quantifier.

Table 6-7. *Quantifiers That Can Be Used for Repeated Matches of Atoms*

Quantifier	Meaning
atom`*`	Greedily matches *atom* zero or more times
atom`+`	Greedily matches *atom* one or more times
atom`?`	Greedily matches *atom* zero or one time
atom`{i}`	Greedily matches *atom* exactly `i` times
atom`{i,}`	Greedily matches *atom* `i` or more times
atom`{i,j}`	Greedily matches *atom* between `i` and `j` times

Numeric Conversions

The Standard Library offers numerous ways to convert between strings and numbers. In Chapter 5, you already encountered two: `std::stringstream` and `sprintf()` / `sscanf()`. And in this chapter, you saw two more: the `std::num_put` and `num_get` locale facets, and of course the regular expression library.

All of these options, however, are somewhat verbose and cumbersome if all you want is to quickly convert a single string or number. The `<string>` header offers functions to perform one-off conversions using a single, straightforward expression.

Another issue with the mechanisms seen so far is that none of them are performant enough for efficiently converting many values in bulk. This was addressed in C++17 by a new header, `<charconv>`, that offers high-performance numeric conversion functions.

In this section, we discuss both sets of conversion functions in turn, starting with those offered by `<string>` and ending with those of `<charconv>`.

■ **Note** The <cstdlib>, <cwchar>, and <cinttypes> headers of the C Standard Library contain even more functions that convert C-style strings and character buffers to numbers: ato[i|l|ll|f](), strto[l|ll|ul|ull|f|d|ld|imax|umax](), and wcsto[l|ll|ul|ull|f|d|ld|imax|umax](). There is little reason to use any of these in C++ though. If convenience is what you need, use either their equivalents defined in the <string> header or the streams seen earlier. And if performance is crucial, you should consider the functions of <charconv>.

Convenient Conversion Functions **<string>**

Parsing Integers

To parse various types of integral numbers represented by a given (w)string, a series of non-member functions of the following form is provided by the <string> header:

```
int stoi(const (w)string&, size_t* index = nullptr, int base = 10);
```

The following variants exist: stoi(), stol(), stoll(), stoul(), and stoull(), where i stands for int, l for long, and u for unsigned. These functions skip all leading whitespace characters, after which as many characters are parsed as allowed by the syntax determined by the base (explained shortly). It is allowed to pass a string that starts with a number, followed by additional characters. If an index pointer is provided, it receives the index of the first character that is not converted.

What constitutes a valid integer string is defined by the current C locale, but it includes at least all strings consisting of the following parts (in the given order):

- An optional + or - sign. A minus sign is allowed even for unsigned target types. It then has the same semantics as the unary minus operator for unsigned integers in C++.
- An optional prefix 0 (zero) if base is either 8 or 0.
- An optional prefix 0x or 0X (zero-x) if base is 16 or 0.
- A nonempty sequence of *digits*. The set of valid digits is determined by the given base, as explained next.

If base is between 2 and 10 (both inclusive), valid digits range from 0 to base-1. If base is between 11 and 36, letters from the range a/A to z/Z are added to the set of valid digits (capital letters are always equivalent to lowercase letters). For the commonly used base 16, for instance, the valid digits are 0–9, a–f, and A–F. base may not be higher than 36.

You can also use 0 (zero) as base. The actual base used for parsing is then determined by the presence of a prefix in the input string (if any): if the string begins with 0x or 0X (zero-x), base 16 is used; otherwise, if it begins with a 0 (zero), base 8 is used; finally, if no prefix is found, base 10 is used.

Parsing Floating-Point Numbers

`stof()`, `stod()`, and `stold()` convert a given string to a `float`, `double`, and `long double`, respectively. They all have the following form:

```
float stof(const (w)string&, size_t* index = nullptr);
```

These conversion functions again consume any leading whitespace and output index of the first unconverted character to `*index` if `index` is not `nullptr`. Valid strings include those consisting of the following parts (additional strings may be accepted based on the active C locale):

- An optional `+` or `-` sign.
- An optional prefix `0x` or `0X` (zero-x) indicating the start of a *hexadecimal* floating-point expression. Without such a prefix, a *decimal* floating-point expression is expected.
- A nonempty sequence of *digits*, optionally containing a *decimal point character*. For a hexadecimal floating-point expression, hexadecimal digits can be used (`0–9`, `a–f`, and `A–F`); otherwise only decimal digits are allowed (`0–9`). The decimal point character is determined by the active C locale.
- An optional *exponent sequence*. For a decimal floating-point expression, this is the letter `e` or `E`, followed by an optional `+` or `-` sign, followed by a nonempty sequence of decimal digits. For the hexadecimal case, `p` or `P` is expected instead of `e` or `E`.

Other valid input strings include `INF`, `INFINITY`, `NAN`, and `NAN(...)` for infinity and (quiet) not-a-number values, all optionally preceded with a `+` or `-` sign, and all case-insensitive. Which character sequences are supported between the parentheses of `NAN(...)` is implementation-defined.

Error Reporting and Number Formatting

All string-to-numeric conversion functions of `<string>` throw an exception of type `std::invalid_argument` if passed a string that does not represent a valid number, and a `std::out_of_range` exception if the target type cannot represent the result.

To perform the opposite conversion and convert from numerical types to a `(w)string`, you can use the overloaded functions `std::to_(w)string()`. Overloads exist for `int`, `unsigned`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, `double`, and `long double` arguments. The returned value is a `std::(w)string`.

High-Performance Conversion Functions C++17 <charconv>

The by far most efficient functions to convert between strings and numbers are those offered by the <charconv> header: std::from_chars() and to_chars(). These are their interfaces (overloads are available for all fundamental *Integer* and *Float* types):

```
from_chars_result from_chars(char* b, char* e, Integer& v, int base = 10)
from_chars_result from_chars(char* b, char* e, Float& v,
                             chars_format format = chars_format::general)

to_chars_result to_chars(char* beg, char* end, Integer val, int base = 10)
to_chars_result to_chars(char* beg, char* end, Float val
                         [, chars_format format][, int precision])
```

Both std::from_chars_result and to_chars_result are structs with these two members (in the given order):

- A const char* pointer named ptr
- An error code named ec of type std::errc, an enum class defined in <system_error> (see Chapter 8)

Unless stated otherwise, ptr points to one past the last character read or written. Upon success, ec equals zero. Upon failure, it has one of the following values:

- errc::invalid_argument if from_chars() fails because it did not find a valid number. ptr then equals b.
- errc::result_out_of_range if from_chars() parses a number that cannot be represented by its output parameter v.
- errc::value_too_large if the output range of to_chars() is not large enough to represent the number.

std::chars_format is a bitmask enumeration type with elements scientific, fixed, and hex, and the default value general that is defined as fixed | scientific.

For from_chars(), the given format affects the strings that can be parsed: if scientific is set but not fixed, the exponent part is no longer optional. Conversely, when fixed is set without scientific, the exponent part may not appear. Whenever hex is set, a prefix 0x is assumed.

For to_chars() the format parameter has the following effect: slightly simplified, to_chars() by default results in the same output as printf() (see Chapter 5), with either %f or %e, whichever results in the shorter string, under the restriction that the result can be parsed again exactly using from_chars(). With a format equal to fixed, however, conversion specifier %f is used instead. Similarly, %e is used with scientific, %a with hex, and %g with general.

The reason these functions can be more efficient is in part because of the following limitations compared to the conversion functions of `<string>`:

- They are independent of the active locale.
- `0` (zero) is not supported as `base`.
- The `+` sign is not allowed in the input string, and the `-` sign is only allowed if the target type is signed.
- No `0x` or `0X` prefix is allowed to appear.

■ **Note** At the time of writing, not all compilers (fully) support these high-performance numeric conversion functions yet: of the compilers we verified, only Visual Studio has full support (as of 2017 15.8), GCC only supports integral conversions (as of GCC 8), and Clang does not provide the `<charconv>` at all yet. Consult your compiler's documentation for more details.

CHAPTER 7

Concurrency

Threads

<thread>

Launching a New Thread

To run any function pointer, functor, or lambda expression in a new thread of execution, pass it to the constructor of `std::thread`, along with any number of arguments. For example, these two lines are functionally equivalent:

```
std::thread worker1(my_callable, "arg", anotherArg);
std::thread worker2([=] { my_callable("arg", anotherArg); });
```

The given callable with its arguments is invoked in a newly launched thread of execution prior to returning from the `thread`'s constructor.

Both the callable and its arguments must first be copied or moved (e.g., for temporary objects or if `std::move()` is used) to memory accessible to this new thread. Therefore, to pass a reference as an argument, you first have to make it copyable. You can do this by wrapping it using `std::ref()` / `std::cref()`. Or, you can simply use a lambda expression with capture-by-reference instead. Functors and reference wrappers are discussed in detail in Chapter 2.

The `thread` class does not offer any facilities to retrieve the callable's result. On the contrary, its return value is ignored, and `std::terminate()` is called if it raises an uncaught exception (which by default terminates the process: see Chapter 8). Retrieving function results is made easier though using the constructs defined in the `<future>` header, as detailed later in this chapter.

Tip To asynchronously execute a function and retrieve its result later, `std::async()` (defined in `<future>`) is recommended over `thread`. It typically is both easier and more efficient (implementations of `async()` likely use a thread pool). Reserve the use of `std::threads` for longer-running concurrent tasks.

P. Van Weert and M. Gregoire, *C++17 Standard Library Quick Reference*,
https://doi.org/10.1007/978-1-4842-4923-9_7

A Thread's Lifetime

A `std::thread` is said to be *joinable* if it is associated with a thread of execution. This property is queried using `joinable()`. `thread`s initialized with a given callable start out joinable, whereas default-constructed ones start out non-joinable. After that, thread instances can be moved and swapped as expected. Copying `thread` objects, however, is not possible. This ensures that at all times, at most one `thread` instance represents a given thread of execution. A handle to the underlying native thread representation may be obtained through the optional `native_handle()` member.

The two most important facts to remember about `std::thread`s are as follows:

- A `thread` remains joinable even after the thread's callable has finished executing.
- If a `thread` object is still joinable when it is destructed, `std::terminate()` is called from its destructor.

So, to make sure the latter does not happen, always make sure to eventually call one of the following functions on each joinable `thread`:

- `join()`: Blocks until the thread's callable has finished executing.
- `detach()`: Disassociates the `thread` object from the thread of execution which continues running autonomously until the thread's callable finishes. Detaching a `thread` is the only standard way to asynchronously execute code in a fire-and-forget manner.

Here is a basic example of using `join()` (although, as noted earlier already, you should probably use `std::async()` instead for cases such as this: see later):

```
double result;
std::thread worker( [&result] { result = someExpensiveComputation(); } );
// ...
worker.join();
std::cout << result << std::endl; // Safe to use result now (see later)
```

A `std::thread` offers no means to terminate, interrupt, or resume the underlying thread of execution. Stopping the thread's callable or otherwise synchronizing with it must therefore be accomplished using other means, such as mutexes or condition variables, both discussed later in this chapter.

Thread Identifiers

Each active thread has a unique `thread::id`, which offers all operations you typically need for thread identifiers:

- They can be outputted to string streams (e.g., for logging purposes).
- They can be compared using == (e.g., for testing/asserting a function is executed on some specific thread).

- They can be used as keys in both ordered and unordered associative containers: all comparison operators (<, >=, etc.) are defined, as is a specialization of `std::hash`.

If a `std::thread` object is joinable, you can call `get_id()` on it to obtain the identifier of the associated thread. All non-joinable `threads` have an identifier that equals the default-constructed `thread::id`. To get the identifier for the currently active thread, you can call the global `std::this_thread::get_id()` function.

Utility Functions

The static `std::thread::hardware_concurrency()` function returns the number of concurrent threads (or an approximation thereof) supported by the current hardware, or zero if this cannot be determined. This number may be larger than the number of physical cores: if the hardware, for instance, supports simultaneous multithreading (branded by Intel as Hyper-Threading), this will be an even multiple of (typically twice) the number of cores.

In addition to `get_id()`, the `std::this_thread` namespace contains three additional functions to manipulate the current thread of execution:

- `yield()` hints the implementation to reschedule, allowing other active threads to continue their execution.
- `sleep_for(duration)` and `sleep_until(time_point)` suspend the current thread for or until a given time; the timeouts are specified using types from `<chrono>` described in Chapter 2.

Exceptions

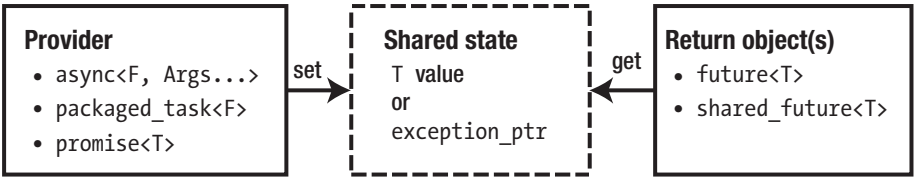
Unless noted here, all functions in `<thread>` are declared `noexcept`. Several `std::thread` members call native system functions to manipulate native threads. If those fail, a `std::system_error` is thrown with one of the following error codes (see Chapter 8 for more information on `system_errors` and error codes):

- `resource_unavailable_try_again` if a new native thread cannot be created in the constructor
- `invalid_argument` if `join()` or `detach()` is called on a non-joinable thread
- `no_such_process` if `join()` or `detach()` is called and the thread is not valid
- `resource_deadlock_would_occur` if `join()` is called on a joinable thread from the corresponding thread of execution

Failure to allocate storage in the constructor may also be reported by throwing an instance of `std::bad_alloc` or a class that derives from `bad_alloc`.

Futures <future>

The <future> header provides facilities to retrieve the result (value or exception) from a function that is being, will be, or has executed, typically in a different thread. Conceptually, a thread-safe communications channel is set up between a single provider and one or more return objects (T may be void or a reference type):



The *shared state* is an internal reference-counted object, shared between a single provider and one or more return objects. The provider asynchronously stores a result into its shared state, which is then said to be *ready*. The only way to acquire this result is through one of the corresponding return objects.

Return Objects

All return objects have a synchronous get() function that blocks until the associated shared state is ready and then either returns the provided value (may be void) or rethrows the provided exception in the calling thread.

To wait until the result is ready without actually retrieving it, use one of the wait functions: wait(), wait_until(time_point), or wait_for(duration). The former waits indefinitely, and the latter two wait no longer than a timeout specified using one of the types defined in <chrono> (Chapter 2).

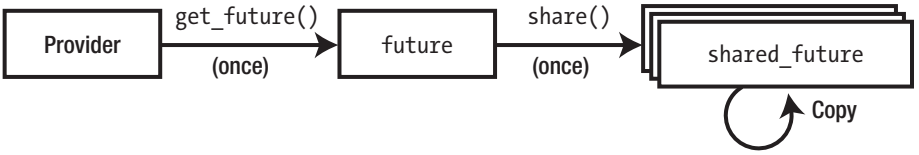
A return object that is associated with a shared state is said to be *valid*. Validity may be checked using valid(). A valid future cannot be constructed directly but must always be obtained from the shared state's single provider.

There are two important limitations with std::futures:

- There can be only one valid future per shared state, just as there can only be one provider. That is, each provider allows the creation of only one future, and futures can never be copied, only moved (futures cannot be swapped either).
- get() can only be called once; that is, calling get() releases the future's reference to the shared state, making the future non-valid. Calling get() again after this throws an exception. Which exceptions are raised and when is summarized at the end of the section.

A shared_future is completely equivalent to a future, but without these two limitations, that is, they can be copied, and get() may be called more than once. A shared_future is obtained by calling share() on a future. This can again be done only

once, because it invalidates the `future`. But once you have a `shared_future`, more can be created by copying it. Here is an overview:



Providers

The `<future>` library offers three different providers: `std::async()`, `packaged_tasks`, and `promises`. This section discusses each in turn. As example workload for asynchronous computations, we use the following greatest common divisor function (known as Euclid's algorithm):

```
int gcd(int x, int y) { return y? gcd(y, x % y) : x; }
```

Async

Calling `std::async()` schedules the asynchronous execution of a given function before returning a `std::future` object that can be used to retrieve the result:

```
std::future<int> answer = std::async(gcd, 123, 6);
// ...
std::cout << answer.get(); // 3 (greatest common divisor of 123 and 6)
```

As with the `std::thread` constructor, virtually any type of function or function object can be used, and both the function and its arguments are moved or copied to their asynchronous execution context.

The result of the function call is put into the shared state as soon as the function is finished executing. If the function throws an exception, the exception is caught and put into the shared state; and if it succeeds, the return value is moved there.

The standard defines additional overrides of `std::async()` that take an instance of `std::launch` as a first argument. Supported values include at least the following `enum` values (implementations are allowed to define more):

- With `std::launch::async`, the function is executed as if in a new thread of execution, although implementations may employ, for example, a thread pool to improve performance.
- With `std::launch::deferred`, the function is not executed until `get()` is called on one of the return objects for this call of `async()`. The function is executed in the first thread that calls `get()`.

These options can be combined using the | operator. For instance, the combination `async | deferred` encourages the implementation to exploit any available concurrency but allows to defer until `get()` is called if there is insufficient concurrency available. This combination is also the default policy used when no explicit launch policy is specified.

There is one important caveat when using a launch policy that includes `async` (i.e., also with the default policy). Conceptually, the thread that executes the asynchronous function is owned by the shared state, and the destructor of the shared state joins with it. As a consequence, the following becomes a *synchronous* execution of `f()`:

```
std::async(f); // Blocks until f() is fully executed!
```

This is because the destruction of the temporary `future` returned by `async()` blocks until `f()` is finished executing (the destruction of the internal shared state joins with the `thread` in which `f()` runs).

■ **Tip** To launch a function without waiting for its result, a.k.a. fire-and-forget, create a `std::thread` object and `detach()` it.

Packaged Tasks

A `packaged_task` is a functor that executes a given function when its `operator()` is called and then stores the result (i.e., a value or an exception) into a shared state. This can, for instance, be used to acquire the result of a function executed by a `std::thread` (recall that the return value of a `thread`'s function is ignored and that `std::terminate()` is called should the function throw an exception):

```
std::packaged_task<int(int, int)> gcd_task(gcd);
auto gcd_future = gcd_task.get_future();    // type: std::future<int>
std::thread worker(std::move(gcd_task), 8, 12);
worker.detach();
// ...
const int four = gcd_future.get();
```

A `packaged_task` constructed with any function, functor, or lambda expression has an associated shared state and is therefore said to be `valid()`; a default-constructed task is not `valid()`. A single `future` to `get()` the function's result can be obtained using `get_future()`.

Like all providers, a `packaged_task` cannot be copied, only moved or swapped. This is why, in the previous example, we had to *move* the task functor to the `thread` (after first obtaining its `future`). It is, however, the only provider that can be used more than once: `reset()` on a valid `packaged_task` releases its old shared state and associates it with a freshly created one. Resetting a non-valid task throws an exception.

There is one additional member function, `make_ready_at_thread_exit()`, which executes the task's function just like `operator()` would, except that it does not make the shared state ready until the calling thread exits. This is done after, and used to avoid race conditions with, the destruction of all thread-local objects:

```
std::packaged_task<int(int, int)> gcd_task(gcd);
std::thread worker([&] { gcd_task.make_ready_at_thread_exit(8, 12); });
worker.detach();
// ...
const int four = gcd_task.get_future().get();
```

Promises

A `promise` is similar to a `future` but represents the input side of the communication channel rather than the output side. Where a future has a blocking `get()` function, a promise offers nonblocking `set_value()` and `set_exception()` functions.

A new `promise` is default-constructed and cannot be copied, only moved or swapped. From each `promise`, a single `future` can be obtained using `get_future()`. If a second is requested, an exception is thrown. Here is an example:

```
std::promise<int> gcd_promise;
std::thread worker([&] { gcd_promise.set_value(gcd(121,22)); });
worker.detach();
// ...
const int eleven = gcd_promise.get_future().get();
```

There is also a second set of member functions to fill in the result: `set_value_at_thread_exit()` and `set_exception_at_thread_exit()`. These again postpone making the shared state ready until the calling thread exits, thus ensuring that this occurs after the destruction of any thread-local objects.

Exceptions

Most functions in the `<future>` header throw an exception if misused. Because the behavior is consistent across all provider and return objects, this single section provides the overview. The following discussion refers to standard exception classes as well as the concepts of error codes and categories, all of which are explained in detail in Chapter 8.

As usual, default and move constructors, move assignment operators, and `swap()` functions are declared `noexcept`, and of course destructors never throw exceptions either. Apart from these, only the `valid()` functions are `noexcept`.

Most other member functions of provider and return objects throw a `std::future_error` in case of an error, a subclass of `std::logic_error`. More similar to a `std::system_error`, though, a `future_error` also has a `code()` member that returns a `std::error_code`, in this case one for which the `category()` equals `std::future_category()` (whose

`name()` equals "`future`"). For `future_errors`, the `value()` of the `error_code` always equals one of the four values of the error code `enum` class `std::future_errc`:

- `broken_promise`, if `get()` is called on a return object for a shared state that was released by its provider—because its destructor, move assignment, or `reset()` function was called—without first making the shared state ready.
- `future_already_retrieved`, if `get_future()` is called twice on the same provider (without a `reset()` for a `packaged_task`).
- `promise_already_satisfied`, if the shared state is made ready multiple times, either by a `set` function or by reexecuting a `packaged_task`.
- `no_state`, if any member except the nonthrowing ones listed earlier is called on a provider without an associated state. For non-`valid()` return objects, implementations are encouraged to do the same.

When using an `async` launch policy, `async()` may throw a `system_error` with error code `resource_unavailable_try_again` if it fails to create a new thread.

Mutual Exclusion <mutex>

Mutexes (short for *mutual exclusion*) are synchronization objects used to prevent or restrict concurrent accesses to shared memory and other resources, such as peripheral devices, network connections, and files.

Mutexes and Locks

Basic usage of a `std::mutex` object `m` is as follows:

```
m.lock();
// ... access to shared resources guarded by m
m.unlock();
```

The `lock()` function blocks until the thread has acquired ownership of a mutex. For a basic `std::mutex` object, only a single thread is granted exclusive ownership at any given time. The intention is that only threads that own a given mutex are allowed to access the resources guarded by it, thus preventing data races. A thread retains this ownership until it releases it by calling `unlock()`. Upon unlocking, another thread that is blocked on the `mutex`, if any, is woken up and granted ownership. The order in which threads are woken up is undefined.

It is critical that any and all successful calls to a lock function are paired with a call to `unlock()`. To ensure this is done in a consistent and exception-safe manner, you should avoid calling these lock and unlock functions directly and use the Resource Acquisition Is Initialization (RAII) idiom instead. For this, the Standard Library offers several lock

classes. One of the simplest, leanest locks is `scoped_lock`, which simply calls `lock()` in its constructor and `unlock()` in its destructor:

```
{  std::scoped_lock lock(m);
   // ... access shared resources guarded by m
}
```

Example

```
int counter = 0;
std::mutex m;
std::vector<std::thread> threads;        // Needs <vector> and <thread>.
for (int t = 0; t < 4; ++t)              // Launch 4 counting threads.
   threads.emplace_back([&] {
      for (int i = 0; i < 500; ++i) {    // Count to 500 in each thread.
         using namespace std::literals::chrono_literals;
         std::this_thread::sleep_for(1ms);
         std::scoped_lock lock(m);
         ++counter;
      }
   });
for (auto& t : threads) { t.join(); }    // Wait for all threads to finish.
std::cout << counter << std::endl;       // 2000
```

The result is 2,000. Removing the `scoped_lock` almost certainly results in a value less than 2,000, unless of course your system cannot execute threads concurrently.

Mutex Types

The Standard Library offers several flavors of mutexes, each with additional capabilities compared to the basic `std::mutex`. More restricted mutex types can typically be implemented more efficiently.

Mutex Type	Recursive	Timeouts	Sharing	Header
`mutex`	No	No	No	`<mutex>`
`recursive_mutex`	Yes	No	No	`<mutex>`
`timed_mutex`	No	Yes	No	`<mutex>`
`recursive_timed_mutex`	Yes	Yes	No	`<mutex>`
`shared_mutex` C++17	No	No	Yes	`<shared_mutex>`
`shared_timed_mutex`	No	Yes	Yes	`<shared_mutex>`

Common Functionality

In addition to the `lock()` and `unlock()` functions explained earlier, all mutex types also support `try_lock()`, a nonblocking version of `lock()`. It returns `true` if ownership can be acquired instantly; otherwise, it returns `false`.[1]

Implementations may also offer a `native_handle()` member, returning a handle to the underlying native object.

None of the mutex types allow copying, moving, or swapping.

Recursion

Recursive mutexes (a.k.a. *reentrant mutexes*) allow lock functions to be called by threads that already own the mutex. When doing so, locking immediately succeeds. Take care though: to release ownership, `unlock()` has to be called once per successful invocation of a lock function. As always, it is therefore best to use RAII lock objects.

For nonrecursive mutex types, the behavior of locking an already-owned mutex is undefined as per the standard, but it may very well lead to a deadlock.

Timeouts

Timed mutexes add two extra lock functions that block until a given timeout: `try_lock_for(duration)` and `try_lock_until(time_point)`. As usual, the timeouts are specified using types defined in `<chrono>`, explained in Chapter 2. Both functions return a Boolean: `true` if ownership of the mutex was acquired successfully or `false` if the specified timeout occurred first.

Sharing Ownership `<shared_mutex>`

Many types of shared resources can safely be accessed concurrently as long as they are not modified. For shared memory, for instance, multiple threads can safely read from a given location, as long as there is no thread writing to it at the same time. Restricting read access to a single thread in such scenarios is overly conservative and may harm performance.

The `<shared_mutex>` header therefore defines mutexes that support shared locking, on top of the exclusive locking scheme they have in common with all other mutex types. Such mutexes are also commonly known as *readers-writers mutexes* or *multiple-readers/single-writers mutexes*.

A thread that intends to modify/write to a resource must acquire *exclusive ownership* of the mutex. This is done using the exact same set of functions or lock objects as used for all mutex types. Threads that only want to inspect/read from a resource, however, can acquire *shared ownership*. The members for acquiring shared ownership are completely analogous to their counterparts for exclusive ownership, except that in their names `lock`

[1] Although normally uncommon, `try_lock()` is allowed to spuriously fail, i.e., return `false` even though the mutex is not owned by any other thread. Take that into account when designing more advanced synchronization scenarios.

is replaced with `lock_shared`; that is, they are named `lock_shared()`, `try_lock_shared_for()`, and so on. Shared ownership is released using `unlock_shared()`.

No exclusive ownership is granted while one or more threads have acquired shared ownership, and vice versa. The standard does not define the order in which ownership is granted or in which threads are unblocked in any way.

The shared locks defined by the standard currently do not support upgrading ownership from shared to exclusive, or downgrading from exclusive to shared, without unlocking first.

Lock Types

std::scoped_lock C++17

`scoped_lock` is a textbook RAII-style class template: by default, it locks a mutex in its constructor and unlocks it in its destructor:

```
std::scoped_lock<std::mutex> lock(m); // locks m, unlocks on destruction
```

Thanks to C++17's class template argument deduction, you can write the previous example shorter as follows:

```
std::scoped_lock lock(m);             // locks m, unlocks on destruction
```

A `scoped_lock` is the only lock type that you can use to safely lock multiple mutexes at once (i.e., without the help of the `std::lock()` function explained later):

```
std::scoped_lock lock(mutex1, mutex2);
```

`std::scoped_lock` offers one more constructor that allows it to acquire one or more mutexes that are already owned by the calling thread. You invoke this constructor by passing the global `std::adopt_lock` constant along as well:

```
std::scoped_lock lock(std::adopt_lock, m);     // unlocks on destruction
```

When adopting a mutex `m`, the lock's constructor no longer calls `m.lock()`. Of course, `m.unlock()` is still called by its destructor as always.

■ **Note** The Standard Library also defines `std::lock_guard`. It is completely equivalent to `std::scoped_lock`, except that it can only be used with a single mutex. If your compiler supports C++17, we recommend you only use `scoped_lock` (to reduce the amount of lock types you use). `scoped_lock` is strictly superior to the old `lock_guard`.

---

std::unique_lock

Although `scoped_lock` is easy and optimally efficient (it uses no additional memory, nor does it generate any branching statements), it is fairly limited in functionality. To facilitate more advanced scenarios, the standard defines `unique_lock`.

The basic usage is the same:

```
std::unique_lock lock(m);   // std::unique_lock<std::mutex> deduced
```

A single `unique_lock` cannot manage the lock for multiple mutexes at once. If `scoped_lock` does not suffice, you can use several `unique_locks` together with the `std::lock()` / `try_lock()` functions discussed later.

That limitation notwithstanding, `unique_lock` does offer several additional, interesting features compared to `scoped_lock`—including these:

- A `unique_lock` can be moved and swapped (but of course not copied).
- You can use `owns_lock()` to check whether the `unique_lock` will unlock upon destruction (`unique_lock` also casts to a Boolean with this value).
- The `mutex()` member returns a pointer to the underlying mutex.
- It has a `release()` function to disassociate it from the underlying mutex without unlocking it (not even in the destructor).

What really sets `unique_lock` apart, though, is that it offers functions to unlock and (re)lock its mutex. Specifically, it supports the exact same set of locking functions as the underlying mutex type: `lock()`, `try_lock()`, and `unlock()`, plus the timed locking functions for timed mutex types.

■ **Caution** Even if the underlying mutex is recursive, the locking functions of `unique_lock` may still be called only once, or an exception will be thrown (which exception is explained at the end of this section).

In addition to the obvious constructor with a given mutex, the `unique_lock` class supports three alternative constructors where you pass an additional constant as the second argument, *after* the mutex (in the corresponding constructor of `scoped_lock`, you pass the `adopt_lock` constant first, *before* the mutex or mutexes):

- `adopt_lock`: Used when the mutex is already owned by the current thread.
- `defer_lock`: Signals not to lock during construction; one of the locking functions may be used to lock the mutex later.
- `try_to_lock`: Tries to lock during construction, but does so without blocking should it fail. `owns_lock()` can be used to check whether it succeeded.

std::shared_lock <shared_mutex>

Most lock types only manage exclusive ownership of mutexes. To reliably manage shared ownership, <shared_mutex> defines std::shared_lock, which is completely equivalent to unique_lock, except that it acquires/releases shared ownership. However, even though they acquire shared ownership, the names of its locking and unlocking members do not contain shared. The lock() function of std::shared_lock, for instance, invokes lock_shared() rather than lock() on the underlying mutex. This is done to ensure that a shared_lock satisfies the requirements for utilities such as std::lock() and std::condition_variable_any, both of which are discussed later.

The following example shows parts of a ConcurrentPerson class that allows true concurrent use of its getters while enforcing that only one thread at a time can modify its state:

```
class ConcurrentPerson {
 public:
   // ...
   std::string GetFirstName() const { // Don't return reference to m_first!
     std::shared_lock lock(m_mutex);
     return m_first;
   }
   void SetFirstName(std::string first) {
     std::unique_lock lock(m_mutex);
     m_first = std::move(first);
   }
   // ...
 private:
    mutable std::shared_mutex m_mutex; // mutable for use in const members
    std::string m_first;
    // ...
};
```

■ **Caution** The preceding class is actually an example of poor concurrent class API design. Let someone be a ConcurrentPerson shared between multiple threads, defined as

```
ConcurrentPerson someone("Jake", "Peralta");
```

Now suppose that one of these threads executes

```
someone.SetFirstName("Raymond");
someone.SetLastName("Holt");
```

Then the other threads will be able to observe the undesired in-between state where `someone` is named "Raymond Peralta". For this and other reasons, it is often best to leave thread synchronization outside of your data classes (a principle called *external synchronization*, vs. the *internal synchronization* employed by `ConcurrentPerson`). Here it would mean to use our original `Person` class and combine it with an external mutex whenever required. That would allow client code to hold an (exclusive) lock until all required setters are called.

Locking Multiple Mutexes

As soon as threads need to acquire ownership of multiple mutexes at the same time, the risk of deadlocks becomes imminent. Different techniques may be employed to prevent such deadlocks: for example, locking the mutexes in all threads in the same order (error-prone), or so-called try-and-back-off schemes. Next to `scoped_lock`, the Standard Library offers the following function template to facilitate this:

```
std::lock(lockable_1, lockable_2, ..., lockable_N);
```

This function blocks until ownership is acquired for all lockable objects passed to it. These can be mutexes (which, after locking, you should transfer to RAII locks using their `adopt_lock` constructors), but also `unique_locks` or `shared_locks` (normally then constructed with `defer_lock`). If all threads use `std::scoped_lock` or `std::lock()` (both mechanisms are compatible), there cannot be any deadlocks.

A nonblocking `std::try_lock()` equivalent of `std::lock()` exists as well. It calls `try_lock()` on all objects in the order they are passed and returns the zero-based index of the first `try_lock()` that fails, or `-1` if they all succeed. If it fails to lock an object, any objects that were locked already are unlocked again first.

Exceptions

Using a mutex before it is fully constructed or after it has been destructed results in undefined behavior. If used properly, only the functions mentioned next may throw an exception.

For mutexes and locks, all `lock()` and `lock_shared()` functions (not the `try_` variants) may throw a `system_error` with one of these error codes (see Chapter 8):

- `operation_not_permitted` if the calling thread has insufficient privileges.
- `resource_deadlock_would_occur` if the implementation detects that a deadlock would occur. Locking a `unique_lock` that already owns the mutex, for instance, triggers this error. In general, however, deadlock detection is only optional: never rely on this!
- `device_or_resource_busy` if it failed to lock because the underlying handle is already locked. For nonrecursive mutexes only of course, but again: detection is only optional.

Any locking functions with timeouts, including the `try_` variants, may also throw timeout-related exceptions.

By extension, both `std::lock()` and the constructors and locking functions of RAII locks may throw the same exceptions as well. Any of the RAII locking functions (*including* the `try_` variants) are guaranteed to throw a `system_error` with `resource_deadlock_would_occur` if `owns_lock()` == `true` (even if the underlying mutex is recursive), and their `unlock()` members will throw one with `operation_not_permitted` if `owns_lock()` == `false`.

If any locking function throws an exception, it is guaranteed that no mutex was locked.

Calling a Function Once

<mutex>

`std::call_once()` is a thread-safe utility function to ensure other functions are called at most once. This is useful, for example, for implementing the lazy initialization idiom:

```
std::once_flag flag;
...
std::call_once(flag, initialise, "a string argument");
```

Or, equivalently (any function or functor may be used):

```
std::call_once(flag, [] { initialise("a string argument"); });
```

Only a single thread that calls `call_once()` with a given instance of `std::once_flag`—a default-constructible, noncopyable, nonmovable helper class—effectively executes the function passed alongside it. Any subsequent calls have no effect. If multiple threads concurrently call `call_once()` with the same flag, all but one is suspended until the one executing the function has finished doing so. Recursively calling `call_once()` with the same flag results in undefined behavior.

Any return value of the function is ignored. If running the function throws an exception, this is thrown in the calling thread, and another thread is allowed to execute with the flag again. If there are threads blocked, one of them is woken up.

Note that `call_once()` is typically more efficient than, and should be preferred at all times over, the error-prone, double-checked locking (anti)pattern.

■ **Tip** Function-local statics (a.k.a. *magic statics*) have exactly the same semantics as `call_once()` but may be implemented even more efficiently. So, although `call_once()` can readily be used for a thread-safe implementation of the singleton design pattern (left as an exercise for you), the use of function-local statics is advised instead:

```
Singleton& getInstance() {
  static Singleton instance;
  return instance;
}
```

Condition Variables <condition_variable>

A *condition variable* is a synchronization primitive that allows threads to wait until some user-specified condition becomes `true`. A condition variable always works in tandem with a mutex. This mutex is also intended to prevent races between checking and setting the condition, which is inherently done by different threads.

Waiting for a Condition

Suppose the following variables are somehow shared between threads:

```
std::mutex m;
std::condition_variable cv;
bool ready = false;
```

Then the archetypal pattern for waiting until `ready` becomes `true` is

```
{  std::unique_lock lock(m);
   while (!ready) cv.wait(lock);
   //... access to other resources guarded by m, if any
}
```

To wait using a `condition_variable`, a thread must first lock the corresponding mutex using a `std::unique_lock<std::mutex>`.[2] As `wait()` blocks the thread, it also unlocks the mutex: this allows other threads to lock the mutex in order to satisfy the shared condition. When a waiting thread is woken up, before returning from `wait()`, it always first locks the mutex again using the `unique_lock`, making it safe to recheck the condition.

■ **Caution** Although threads waiting on a condition variable normally remain blocked until a notification is done on that variable (discussed later), it is also possible (albeit unlikely) for them to wake up spontaneously at any time without notification. These are called *spurious wakeups*. This phenomenon makes it critical to always check the condition in a loop as in the example.

Alternatively, all wait functions have an overload that takes a predicate function as an argument: any function or functor that returns a value that can be evaluated as a Boolean may be used. The loop in the example, for instance, is equivalent to

```
cv.wait(lock, [&]{ return ready; });
```

[2] With `condition_variable`, this exact lock and mutex type must be used. To use other standard types, or any object with public `lock()` and `unlock()` functions, the more general `std::condition_variable_any` class is declared, which is otherwise analogous to `condition_variable`.

There are two sets of additional wait functions that never block longer than a given timeout: `wait_until(time_point)` and `wait_for(duration)`. The timeouts are, as always, expressed using types defined in the `<chrono>` header. The return value of `wait_until()` and `wait_for()` is as follows:

- The versions of the functions without a predicate return a value from the enum class `std::cv_status`: either `timeout` or `no_timeout`.
- The overloads that do take a predicate function return a Boolean: `true` if the predicate returns `true` after a notification, a spurious wakeup, or when the timeout is reached; otherwise, they return `false`.

Notification

Two notification functions are provided: `notify_all()`, which unblocks all threads waiting on a condition variable, and `notify_one()`, which unblocks only a single thread. The order in which multiple waiting threads are woken up is unspecified.

Notification normally occurs because the condition has changed:

```
{  std::scoped_lock lock(m);
   ready = true;
}
cv.notify_all();
```

Note that the notifying thread is not required to own the mutex when calling a notification function. In fact, the first thing any unblocked thread does is attempt to lock the mutex, so releasing ownership prior to notification may actually improve performance.[3]

There is one more notification function, but it is a non-member function and has the following signature:

```
void std::notify_all_at_thread_exit(condition_variable& cv,
                                    unique_lock<mutex> lock);
```

It is to be called while the mutex is already owned by the calling thread through the given `unique_lock`, and while no thread is waiting on the condition variable using a different mutex; otherwise, the behavior is undefined. When called, it schedules the following sequence of operations upon thread exit, after all thread-local objects have been deleted:

```
lock.unlock();
cv.notify_all();
```

[3] Some care must be taken: it introduces a window for race conditions between setting the condition and the notification of waiting threads. In certain cases, notifying while holding the lock may actually lead to more predictable results and avoid subtle races. When in doubt, it is best to not unlock the mutex when notifying, because the performance impact is likely to be minimal.

Exceptions

The constructor of a condition variable may throw a `std::bad_alloc` if insufficient memory is available, or a `std::system_error` with `resource_unavailable_try_again` as an error code if the condition variable cannot be created due to a nonmemory-related resource limitation.

Destructing a condition variable upon which a thread is still waiting results in undefined behavior.

L1 Data Cache Line Size C++17 <new>

Fetching data from main memory is slow, at least when compared to the speed at which modern processors operate. To ameliorate that, processors generally use multiple levels of caches, small memories that can hold smaller amounts of data, but that are much faster than main memory. The fastest (and smallest) of these caches is the L1 cache. The more data a processor finds readily available in its L1 cache, the less cycles it wastes waiting for data to arrive. There are generally multiple L1 caches in a modern multicore processor (one per core).

Data is always fetched into L1 caches in contiguous blocks of fixed size, even if you only need part of that size. These blocks are called *cache lines*. In C++17, two global constants are added to the `<new>` header to query two related properties of the L1 data cache line size (a typical value for both mostly equal constants is 64):

- `std::hardware_constructive_interference_size`: Maximum size in bytes of contiguous memory to promote *true sharing*
- `std::hardware_destructive_interference_size`: Minimum offset in bytes between two objects to avoid *false sharing*

True sharing occurs when data that is frequently needed at the same time gets loaded in the same cache line. Provided you mind the alignment of your `Pod` objects themselves, their `pea` variables can thus be accessed efficiently together:

```
struct Pod {
   int pea_1;
   // Possibly some other data members in between...
   int pea_2;
};
static_assert(sizeof(Pod) <= std::hardware_constructive_interference_size);
```

False sharing occurs when multiple cores access different values in the same cache line, and at least one of them is writing. If one core writes to a variable, entire cache lines may need to be reloaded for the L1 caches of other cores (most cache coherence schemes require this). Suppose that you omit the `alignas()` specifiers in the following example, and that you write to the `fire` variable from one thread and to `ice` from another. Then, even though both threads never access the same variable, they risk constantly invalidating the caches of the cores they are running on. All because the `fire` and `ice`

variables are likely loaded in the same L1 cache lines. By adding extra padding between the variables, you avoid such false sharing:

```
struct KeepApart {
   alignas(std::hardware_destructive_interference_size) float fire;
   alignas(std::hardware_destructive_interference_size) int ice;
};
```

Synchronization

Informally, for a single-threaded program, an optimizing implementation (the combination of a compiler, the memory caches, and the processor) is bound by the *as-if* rule. Essentially, in a well-formed program, instructions may be reordered, omitted, invented, and so on, at will, as long as the observable behavior (I/O operations and such) of the program is *as if* the instructions were executed as written.

In a multithreaded program, however, this does not suffice. Without proper synchronization, concurrently accessing shared resources inevitably causes data and other races, even if each individual thread adheres to the as-if rule.

Although a full, formal description of the *memory model* is out of the scope of this Quick Reference, this chapter provides a brief informal introduction to the synchronization constraints imposed by the different constructs, focusing on the practical implications when writing multithreaded programs. We introduce all essential synchronization principles first using mutexes. Recall the following:

```
m.lock();    // acquire fence
// ...       (critical section)
m.unlock();  // release fence
```

First, synchronization constructs introduce constraints on the code reorderings that are allowed *within a single thread of execution*. Locking and unlocking a mutex, for example, inject special instructions, respectively called *acquire* and *release fences*. These instructions tell the implementation (not just the compiler but also all hardware executing the code!) to respect these rules: no code may move *up* an acquire fence or *down* a release fence. Together, this ensures that no code is executed outside the *critical section*, the section between `lock()` and `unlock()`.

Second, fences impose constraints *between different threads of execution*. This can be reasoned about as restrictions on the allowed interleavings of instructions of concurrent threads into a hypothetical single instruction sequence. Releasing ownership of a mutex in one thread, for example, is said to *synchronize with* acquiring it in another: essentially, in any interleaving, the former must occur *before* the latter. Combined with the intrathread constraints explained earlier, this implies that the entire critical section of the former thread is guaranteed to be fully executed before the latter thread enters its critical section.

For condition variables, the synchronization properties are implied by the operations on the corresponding mutexes.

For `std::threads`, the following applies:

- When launching a `thread`, its constructor injects a release fence, which synchronizes with the beginning of the execution of the thread function. This implies that you can write to shared memory (e.g., to initialize it or to pass input) before launching a `thread` and then safely (without extra synchronization) access it from within the thread function.
- Conversely, the end of a `thread`'s function execution synchronizes with the acquire fence inside its `join()` function. This ensures that the joining thread can safely read all shared data written by the thread function.

Finally, for the constructs in the `<future>` header, making the shared state ready through a provider contains a release fence, which synchronizes with the acquire fence inside the `get()` of a return object of the same shared state. So not only can the thread that calls `get()` safely read the result (luckily), but it can also safely read any other values written by the provider. So, a `future<void>`, for example, can be used to wait until a thread has finished asynchronously writing to shared memory. Or a `future<T*>` may point to an entire data structure created by the provider function.

■ **Note** All this may be summarized as follows: the behavior of unsynchronized data races (threads concurrently accessing memory with at least one writing) is undefined. However, as long as you consistently use the synchronization constructs provided by the Standard Library, your program will behave exactly as expected.

Atomic Operations `<atomic>`

First and foremost, the `<atomic>` header defines two types of *atomic variables*, special variables whose operations are *atomic or data-race-free*: `std::atomic<T>` and `std::atomic_flag`. In addition, it provides some low-level functions to explicitly introduce fences, as explained at the end of this section.

Atomic Variables

Variables of the `std::atomic<T>` type mostly behave like regular `T` variables—thanks to the obvious constructors and assignment and cast operators—offering a restricted set of fine-grained atomic operations with specific memory consistency properties. More details follow shortly, but first we introduce the template specializations of `atomic<T>`.

Template Specializations and Type Aliases

The `atomic<T>` template may be used at least with any trivially copyable[4] type `T`, and specializations are defined for all integral types `T` (including `bool`), and pointer types `T*`. The latter two offer additional operations, as described later.

For the Boolean and integral specializations, convenience type aliases are defined. For `std::atomic<xxx>`, these mostly equal `std::atomic_xxx`. Specifically, this is true for `xxx` equal to `bool`, `char`, `char16_t`, `char32_t`, `wchar_t`, `short`, `int`, `long`, or any integral type defined in `<cstdint>` (see Chapter 1). For the remaining integral types, the alias abbreviates the first words of the `xxx` type:

Type Alias	xxx	Type Alias	xxx
`std::atomic_schar`	`signed char`	`std::atomic_ulong`	`unsigned long`
`std::atomic_uchar`	`unsigned char`	`std::atomic_llong`	`long long`
`std::atomic_ushort`	`unsigned short`	`std::atomic_ullong`	`unsigned long long`
`std::atomic_uint`	`unsigned int`		

Basic Atomic Operations

The default constructor of an `atomic<T>` variable behaves exactly like the declaration of a regular `T` variable, that is, it generally *does not* initialize the value; only static or thread-local `atomic` variables are zero-initialized:

```
std::atomic_int atom;       // Uninitialized!
```

A constructor to initialize with a given `T` value is present as well. This initialization is not atomic, though: any operation concurrent with a variable's construction results in a data race:

```
std::atomic_int atom{ -123 };         // -123
```

All `atomic<T>` types have both an assignment operator accepting a `T` value and a cast operator to convert to `T` and can therefore be used like regular `T` variables:

```
atom = 456;
std::cout << atom << std::endl;        // 456
```

[4] A *trivially copyable* type has no nontrivial copy/move constructor/assignment, no virtual functions or bases, and a trivial destructor. Essentially, these are the types that can safely be bitwise copied (e.g., using `memcpy()`).

Equivalent to these operators are the `store()` and `load()` members. The last two lines of the previous code snippet, for example, can also be written as

```
atom.store(123);
std::cout << atom.load() << std::endl; // 123
```

Either way, these operations are atomic or, in other words, *data-race-free*. That is, if one thread concurrently stores a value into an atomic variable while another is loading from it, the latter sees either the old value from prior to the store or the newly stored value, but nothing in between (no half-written values). Or, in technical speak, there are no *torn reads*. Similarly, when two threads concurrently each store a value, one of these values is fully stored; there are never *torn writes*. With regular variables, such scenarios are data races and therefore result in undefined behavior, including the possibility of torn reads and writes.

Atomic Exchange of Values

All atomic variables also offer a few less obvious atomic operations, `exchange()` and `compare_exchanges`. These member functions behave as if implemented as follows:

```
T exchange(T newVal) {
  T oldVal = load();
  store(newVal);
  return oldVal;
}
```

```
bool compare_exchange(T& expected, T desired) {
  if (load() == expected) {
      store(desired); return true;
  } else {
      expected = load(); return false;
  }
}
```

This is not how they are actually implemented, though, as all exchange operations are again atomic. That is, they (conditionally) exchange the value in such a way that no thread may concurrently store another value during the exchange or experience a torn read.

Also, there is no actual member named `compare_exchange`. Instead, there are two different variants: `compare_exchange_weak()` and `compare_exchange_strong()`. The only (subtle) difference is that the former is allowed to spuriously fail, that is, sporadically return `false` even when a valid exchange could be done. This "weak" variant may be slightly faster than the "strong" variant but is intended to be used only in a loop. The latter is intended to be used as a stand-alone statement.

The `exchange()` and `compare_exchange` operations are key building blocks in the implementation of *lock-free data structures*: thread-safe data structures that do not use blocking mutexes. This is an advanced topic, best left to experts. Still, a classical example is adding a new node to the beginning of a singly linked list (why this code excerpt usually does not use any mutexes is explained shortly):

```
Node* head = m_atomic_head.load();    // m_atomic_head is an atomic<Node*>
Node* newNode = new Node(value, head);
while (!m_atomic_head.compare_exchange_weak(head, newNode))
   newNode->next = head;
```

Lock-Free Atomic Operations

Unless otherwise noted, all member functions of `std::atomic<T>` are atomic, that is, data-race-free. For small base types `T`, including fundamental types and pointers, most compilers generate one or a few special hardware instructions that guarantee atomicity (most current CPUs support this). For larger base types `T`, `std::atomic<T>` mostly resorts to more expensive mutex-like constructs to accomplish atomicity. You can check whether the operations for a given `atomic<>` object will require locks using its `lock_free()` method:

```
std::atomic<float> fundamental;
std::atomic<std::array<int, 2>> small_type;
std::atomic<std::array<int, 100>> large_type;
std::cout << std::boolalpha                    // Likely results:
          << fundamental.is_lock_free() << '\n'  // true
          << small_type.is_lock_free() << '\n'   // true
          << large_type.is_lock_free() << '\n';  // false
```

Starting with C++17, you can also check this statically using the static member constant `std::atomic<T>::is_always_lock_free` (a Boolean).

Take care: although atomic variables ensure that loads and stores are atomic, this does not make the operations on the underlying object atomic. In the following example, if another thread concurrently calls `GetLastName()` on the `person` object, then there is a data race with `SetLastName()`:

```
std::atomic<Person*> person( new Person("Phil") );  // non-atomic init.
// ... (share references to person with other threads)
person = new Person("Claire");         // atomic store
person.load()->SetLastName("Dunphy");  // atomic load, non-atomic setter!
```

■ **Note** This even holds for types such as `std::atomic<float>`: while store and load are atomic, you cannot atomically apply arithmetic operations on the underlying `float` (although this may change in C++20). The Standard Library only defines certain atomic arithmetic operations for integral and pointer types, as we explain in the next section.

Atomic Operations for Integral and Pointer Types

Certain template specializations offer additional operators that update the variable without the possibility of a data race. The selection is based on which atomic instructions current hardware generally supports (no multiplication, for instance):

- Atomic integral variables: ++, --, +=, -=, &=, |=, ^=
- Atomic pointer variables: ++, --, +=, -=

Both pre- and postfix versions of ++ and -- are supported. For the other operators, equivalent non-operator members are again available as well: respectively, `fetch_add()`, `fetch_sub()`, `fetch_and()`, `fetch_or()`, and `fetch_xor()`.

Synchronization

In addition to atomicity, a lesser-known property of atomic variables is that they offer the same kind of synchronization guarantees as, for example, mutexes or threads. Specifically, all operations that write to a variable (`store()`, exchanges, `fetch_xxx`(), etc.) contain release fences that synchronize with the acquire fences in operations that read from the same variable (`load()`, exchanges, `fetch_xxx()`, etc.). This enables the following idiom, which initializes a potentially complex object or data structure before storing it in a shared atomic variable:

```
std::atomic<Person*> atomic_person(nullptr);
// ... (share references to atomic_person with other threads)
auto person = new Person();
person->SetFirstName("Jay");
person->SetLastName("Pritchett");
atomic_person = person;        // atomic store + release fence!
```

Any thread that loads the pointer to the new object (a `Person` in this example) can safely read all other memory it points to as well (e.g., the name strings), as long as this was completely written prior to the release fence.

All atomic operations (except the operators, of course) accept an extra, optional `std::memory_order` parameter (or parameters), allowing the caller to fine-tune the memory order constraints. Possible values are `memory_order_relaxed`, `memory_order_consume`, `memory_order_acquire`, `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst` (the default). The first option, `memory_order_relaxed`, for instance, denotes that the operation simply has to be atomic and that no further memory order constraints are required. The often subtle differences between the other options fall outside the scope of this book. Unless you are an expert, our recommendation is that you stick with the default values. Otherwise, you risk introducing subtle bugs.

Atomic Flags

The `std::atomic_flag` is a simple, guaranteed lock-free, atomic, Boolean-like type. It can only be default-constructed and cannot be copied, moved, or swapped. It is not specified to which state (`true` or `false`) the default constructor initializes the flag. The only other initialization that is guaranteed to work is this *exact* expression:

```
std::atomic_flag guard = ATOMIC_FLAG_INIT;    // Initilizes guard to false
```

An `atomic_flag` offers only two more members (note that there is not even a function or cast operator to get the current state of the flag):

- `void clear()`: Atomically sets the flag to `false`
- `bool test_and_set()`: Atomically sets the flag to `true` while returning its previous value

Both functions have synchronization properties similar to `atomic_bool`s and again accept an optional `std::memory_order` parameter as well.

Non-member Functions and Macros

For compatibility with C, `<atomic>` defines non-member counterparts or C macros for all member functions of `std::atomic<T>` and `std::atomic_flag`: `atomic_init()`, `atomic_load()`, `atomic_fetch_add()`, `atomic_flag_test_and_set()`, and so on. As a C++ programmer, you normally never need any of these: simply use the classes' member functions.

Fences

The `<atomic>` header also provides two functions to explicitly create acquire and/or release fences: `std::atomic_thread_fence()` and `std::atomic_signal_fence()`. The concept of fences is as explained earlier in this chapter. Both take a `std::memory_order` argument to specify the desired fence type: `memory_order_release` for a release fence, either `memory_order_acquire` or `memory_order_consume` for an acquire fence, and `memory_order_acq_rel` and `memory_order_seq_cst` for fences that are both acquire and release fences, with the latter option denoting the fence has to be the sequentially consistent variant (the difference in their semantics falls outside the scope of this book). A fence with `memory_order_relaxed` has no effect.

The difference between the two functions is that the latter only restricts reorderings between a thread and a signal handler executed in the same thread. The latter only constrains the compiler but does not inject any instructions to constrain the hardware (memory caches and CPU).

■ **Caution** Using explicit fences is discouraged: atomic variables or other synchronization constructs have more interesting synchronization properties and should generally be preferred instead.

CHAPTER 8

Diagnostics

Assertions <cassert>

Assertions are Boolean expressions that are expected to be `true` at a given point in the code. The `assert` macro of `<cassert>` is defined similar to

```
#ifdef NDEBUG
 #define assert(_)
#else
 #define assert(CONDITION) \
   if (!(CONDITION)) { print_msg(__FILE__, __LINE__, ...); std::abort(); }
#endif
```

If an assertion fails, a diagnostic message is written to the standard error output, and `std::abort()` is called which terminates the application without performing any cleanup. The diagnostic message typically includes the filename, the line number, and the conditional expression that triggered the assertion. While debugging an application, certain IDEs give you the option to continue the execution if an assertion fails. Common practice is to use assertions as a debugging aid, and to define `NDEBUG` when building a release build of your application, turning `assert`s into no-operations.

Assertions are generally used to check invariants, such as loop invariants, pre- and postconditions in functions, and so on. One example is parameter validation:

```
void foo(const char* msg) { assert(msg != nullptr); }  // or: assert(msg);
int main() {
   foo("Test");  // OK
   foo(nullptr); // Triggers the assertion.
}
```

A possible output of this program is

```
Assertion failed: msg != nullptr, file d:\Test\Test.cpp, line 13
```


P. Van Weert and M. Gregoire, *C++17 Standard Library Quick Reference*,
https://doi.org/10.1007/978-1-4842-4923-9_8

■ **Caution** Make sure that the condition you provide to `assert()` does not have any side effects that are required for the proper execution of your program, because this expression does not get evaluated if NDEBUG is defined (e.g., for a release build).

Exceptions

<exception>, <stdexcept>

`std::exception`, defined in `<exception>`, is not intended to be thrown itself, but instead serves as a base class for all exceptions defined by the Standard Library and can serve as a base class for your own. Figure 8-1 outlines all standard exceptions.

An `exception` can be copied and offers a `what()` method that returns a string representation of the error. This function is virtual and should be overridden. The return type is `const char*`, but the character encoding is not specified (i.e., Unicode strings encoded as UTF-8 could be used, for instance; cf. Chapter 6).

The exceptions defined in `<stdexcept>` are the only standard exceptions that are intended to be thrown by application code. As a rule, `logic_error`s represent avoidable errors in the program's logic, while `runtime_error`s are caused by less predictable events beyond the scope of the program. `logic_error`, `runtime_error`, and most of their subclasses must be passed a `std::string` or `const char*` pointer upon construction, which shall be returned by `what()` afterward. There is thus no need to further override `what()` when deriving from one of these classes.

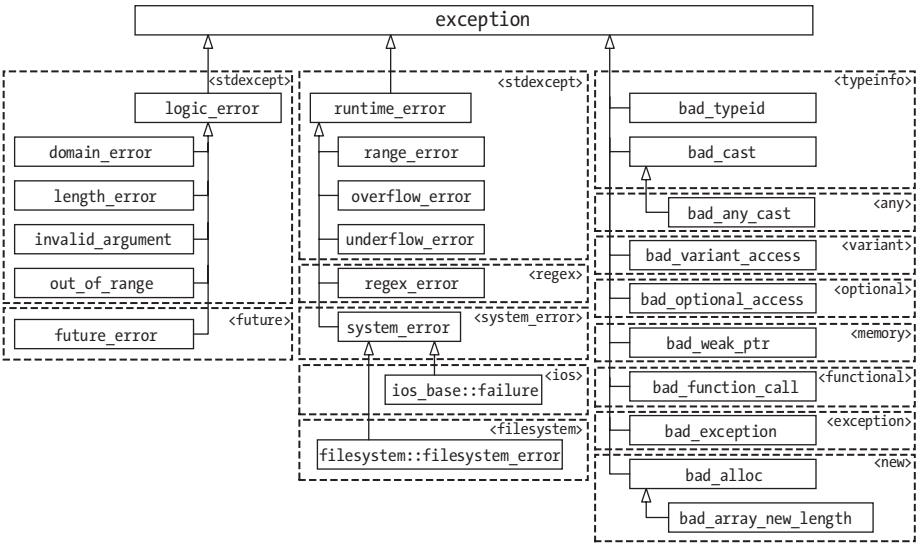


Figure 8-1. *The C++ Standard Library exception hierarchy*

Exception Pointers <exception>

The <exception> header provides std::exception_ptr, an unspecified pointer-like type, used to store and transfer caught exceptions, even without knowing the concrete exception type. In fact, an exception_ptr can point to a value of any type, that is, not just a std::exception. It can just as well point to a custom exception class, an integer, a string, and so on. Any pointed-to value stays valid while there is at least one exception_ptr still referring to it (i.e., a reference-counted smart pointer may be used to implement exception_ptr).

There are a couple of functions defined in <exception> to work with exception pointers:

```
exception_ptr std::current_exception() noexcept
```

Creates and returns an exception_ptr that refers to the exception currently in flight (remember: this can be of any type) when called from inside a catch() block, either directly or indirectly (i.e., a catch() block may call, e.g., a helper function to handle an exception). The returned exception_ptr refers to a null value if called when no exception is being handled.

```
template<typename T>
exception_ptr std::make_exception_ptr(T t) noexcept
```

Creates and returns an exception_ptr that points to a copy of t.

```
[[noreturn]] void std::rethrow_exception(exception_ptr e)
```

Rethrows the exception to which the given exception_ptr points. This is the only way to obtain the object pointed to by an exception_ptr. An exception_ptr cannot be dereferenced, nor is there a getter function.

Once created, exception_ptrs can be copied, compared, and in particular assigned and compared with nullptr. This makes them useful to store and move exceptions around and to test later whether an exception has occurred. For this, an exception_ptr is also convertible to a Boolean: true if it points to an exception, false if it is a null pointer. Default-constructed instances are equivalent to nullptr.

Exception pointers can be used, for example, to transfer exceptions from a worker thread to the main thread (note that this is essentially what the utilities of <future> discussed in the previous chapter implicitly do for you as well):

```
std::exception_ptr threadException;
std::thread t([&threadException] {               // Needs <thread>
  try {
    throw std::invalid_argument("Test");         // In worker thread
  } catch (...) {
    threadException = std::current_exception();  // Store exception
  }
});

t.join();              // Wait for thread to finish.

if (threadException) { // In main thread: handle exception if there is one.
  try {
    std::rethrow_exception(threadException);
  } catch (const std::exception& caughtException) {
    std::cout << "Caught from thread: " << caughtException.what() << '\n';
  }
}
```

Nested Exceptions <exception>

The <exception> header also offers facilities to work with *nested exceptions*. They allow you to wrap a caught exception in another one, for instance, to augment it with extra context information or to convert it to a more suitable exception for your application. `std::nested_exception` is a copyable mixin[1] class whose default constructor captures `current_exception()` and stores it. This nested exception can be retrieved as an `exception_ptr` with `nested_ptr()`, or by using `rethrow_nested()` which rethrows it. Take care though: `std::terminate()` is called when `rethrow_nested()` is called without any stored exception. It is therefore generally recommended to not use `nested_exception` directly, but to use the following helper functions instead:

[1] A mixin is a class that provides some functionality to add to other classes (in this case, the capability of storing a pointer to a nested exception and some related functions). In C++, mixins are generally implemented through multiple inheritance.

`template<typename T> [[noreturn]] void std::throw_with_nested(T&& t)`

Throws an undefined type deriving from both `std::nested_exception` and `T` (with reference qualifiers stripped) which can be handled using a regular `catch (const T&)` expression, ignoring the nested exception altogether. Being a `std::nested_exception` as well though, it also contains the result of `std::current_exception()`, which may optionally be retrieved and handled.

`template<typename T> void std::rethrow_if_nested(const T& t)`

If `t` derives from `nested_exception`, then calls `rethrow_nested()` on it, otherwise does nothing.

The following example demonstrates nested exceptions:

```
void execute_helper() {
   throw std::range_error("Out-of-range error in execute_helper()");
}
void execute() {
   try { execute_helper(); }
   catch (...) {
      std::throw_with_nested(std::runtime_error("Caught in execute()"));
   }
}
void print(const std::exception& exc) {
   std::cout << "Exception: " << exc.what() << std::endl;
   try { std::rethrow_if_nested(exc); }
   catch (const std::exception& e) {
      std::cout << "   Nested ";
      print(e);
   }
}
int main() {
   try { execute(); }
   catch (const std::exception& e) { print(e); }
}
```

The output of this piece of code is as follows:

```
Exception: Caught in execute()
   Nested Exception: Out-of-range error in execute_helper()
```

System Errors <system_error>

Errors from the operating system or other low-level APIs are called *system errors*. These are handled by classes and functions defined in the <system_error> header:

- error_code: Generally wraps a platform-specific error code (an int), although for some categories the error codes are defined by the standard (cf. Table 8-1).
- error_condition: Wraps a portable, platform-independent error condition (an int). The enum class std::errc lists the built-in conditions. They correspond to the standard POSIX error codes, defined also as macros in <cerrno>. See Table 8-2 at the end of this chapter.
- error_category: A base class for specific error categories. Error codes and conditions belong to a category. Specific category singleton objects are responsible for converting between both numberings.
- system_error: An exception class (cf. Figure 8-1) with an extra code() method returning an error_code.

Table 8-1. *Available Error Category Functions and Corresponding Error Condition and Error Code Enum Classes*

Singleton Function	Error Conditions	Error Codes	Header
generic_category()	std::errc		<system_error>
system_category()			<system_error>
iostream_category()		std::io_errc	<ios>
future_category()		std::future_errc	<future>

Next to a numeric value, both error_code and error_condition objects hold a reference to their error_category. Within one category, a number is unique, but the same number may be used by different categories.

While all this may seem fairly complicated, their main use is rather straightforward. To compare a given error code, for instance, that from a caught system_error exception, with either an error condition or code, the == and != operators can simply be used. For instance:

```
if (systemError.code() == std::errc::argument_out_of_domain)
   ...
```

■ **Note** Working with std::ios_base::failure (Chapter 5) and future_error (Chapter 7) is analogous. They also have a code() method returning an error_code that can be compared with known code values (cf. Table 8-1) using == and !=.

std::error_category

The different `std::error_category` instances are implemented as singletons, that is, there is only one global, noncopyable instance per category. A number of predefined categories exist, obtainable from the global functions listed in Table 8-1.

An `std::error_category` has the following methods:

Method	Description
`const char* name()`	Returns the category's name.
`string message(int)`	Returns an explanatory `string` for a given error condition value.
`error_condition default_error_condition(int)`	Converts a given error code value to a portable `error_condition`.
`bool equivalent(int, error_condition&)` `bool equivalent(error_code&, int)`	Compares error codes (given as an `int` or as an `error_code`) with portable conditions (given as an `error_condition` or as an `int`). However, it's easier to use the `==` and `!=` operators shown earlier instead.

std::error_code

`std::error_code` encapsulates an error code value and an `error_category`. There are three constructors:

- A default one setting the error code to `0` (this conventionally represents 'no error') and associates it with `system_category`.
- One accepting an error code `int` and an `error_category`.
- One constructing an `error_code` from an *error code enumeration value* `e` by calling `std::make_error_code(e)`. The parameter type must be an *error code enumeration type*, an enumeration type for which the `std::is_error_code_enum` type trait has a value of `true` (see Chapter 2 for type traits). This automatically sets the correct category as well. The enum classes for the standard categories are shown in Table 8-1.

To raise your own `std::system_error`, you have to provide an `error_code` which can be created with one of its constructors or with `make_error_code()`. For example:

```
throw std::system_error(std::make_error_code(std::errc::invalid_argument),
   "Now what am I to do with that argument?");  // optional what() message
```

`std::error_code` provides the following methods:

Method	Description
`assign(int, error_category&)`	Assigns the given error code and category to this `error_code`
`operator=()`	Uses `std::make_error_code()` to assign a given error code enumeration value to this `error_code`
`clear()`	Sets the error code to 0 and the category to `system_category` to represent no error
`int value()` `error_category& category()`	Returns the error value/associated category
`error_condition default_error_condition()`	Calls `category().default_error_condition(value())`, returning the corresponding portable error condition
`string message()`	Calls `category().message(value())`
`operator bool()`	Returns `true` if the error code is not 0

std::error_condition

The `std::error_condition` class encapsulates a portable condition code and the associated error category. This class has a similar set of constructors and methods as `error_code`, except

- It does not have a `default_error_condition()` method.
- Error *condition* enumerations are used instead of error *code* enumerations, that is, those enum types for which the `is_error_condition_enum` type trait has a value of `true`.
- Members that use `std::make_error_code()`, use `std::make_error_condition()` instead.

C Error Numbers <cerrno>

The `<cerrno>` header defines `errno`, a macro that expands to a value equivalent to `int&`. Functions can set the value of `errno` to a specific error value to signal an error. A separate `errno` is provided per thread of execution. Setting `errno` is very common for functions from the C headers. The C++ libraries mostly throw exceptions upon failure, although some set `errno` as well (e.g., `std::string`-to-numeric conversions). Table 8-2 lists the macros with default POSIX error numbers defined by `<cerrno>`.

If you want to use errno to detect errors in functions that use errno to report errors, then you have to make sure to set errno to 0 before calling the function, as is done in this example (needs <cmath>)[2]:

```
errno = 0;                          // Reset errno to 0!
auto result = std::exp(100000);     // Causes an overflow error.
// Convert the errno error code to an error_code instance.
std::error_code errorCode(errno, std::generic_category());
std::error_condition okCondition;  // Default constructor creates
                                   // a no-error condition.
if (errorCode != okCondition)      // Check for an error.
   std::cerr << "Error: " << errorCode.message() << std::endl;
```

The output depends on your platform, but can be something as follows:

```
Error: result out of range
```

For completeness, we show two alternative ways of converting the current errno value to an error string. They use strerror() from <cstring> (take care though: this function is not thread-safe!) and std::perror() from <cstdio>, respectively. The following two lines print a similar message as the preceding code:

```
std::cerr << "Error: " << std::strerror(errno) << std::endl;
std::perror("Error");     //Prefix string is non-optional
```

Failure Handling

<exception>

std::uncaught_exceptions() C++17

You can use uncaught_exceptions() to determine whether one or more exceptions are currently in flight that have not been caught yet—in other words, detect that stack unwinding is in progress. It returns the number of thrown exceptions that have not been caught yet.

The prime use case for this function is to figure out whether a destructor of a stack-allocated object is being executed as a result of stack unwinding or not. To do this, you should first store the result of uncaught_exceptions() in one of its member variables

[2] std::exp() only sets errno for implementations where math_errhandling defined in <cmath> contains MATH_ERRNO: see Chapter 1. This appears to be mostly the case though.

during initialization. In its destructor, you then call `uncaught_exceptions()` again, and compare its result with that member. Like so:

```
class Guard {
   const int exception_count = std::uncaught_exceptions();
public:
   ~Guard() {
      std::cout << (exception_count == std::uncaught_exceptions()
         ? "~Guard() invoked normally\n"
         : "~Guard() invoked during stack unwinding\n");
   }
};
```

This pattern can be used for so-called *scope guards* that conditionally run some piece of code in their destructor. For instance, to roll back a failed (and potentially half-finished) operation when an exception is thrown in the same scope. Or, similarly, to commit an operation, but then only if no exception occurs.

■ **Note** The `uncaught_exception()` function, that is, singular instead of plural, returns a Boolean, `true` if any exception has been thrown that is not caught yet, `false` otherwise. It was deprecated in C++17 in favor of `uncaught_exceptions()` because there usually is no reason or safe way to use it.

---

std::terminate()

If exception handling fails for any reason, for example, an exception is thrown but never caught, then the runtime calls `std::terminate()` which calls the so-called *terminate handler*. The default handler calls `std::abort()`, which in turn aborts the application without performing any further cleanup. The active terminate handler is managed using the following functions from `<exception>`, where `std::terminate_handler` is a function pointer type and must point to a `void` function without arguments:

```
std::terminate_handler std::set_terminate(std::terminate_handler) noexcept
std::terminate_handler std::get_terminate() noexcept
```

One use case for a custom terminate handler is to automatically generate a process dump when `std::terminate()` is called. Having a dump file to analyze aids tremendously in tracking down a bug that triggered a process to `terminate()`. One should consider setting this up for any professional application.

Table 8-2. *`std::errc` Error Condition Values and Corresponding `<cerrno>` Macros*

`std::errc` enum **Value**	**`<cerrno>` Macro**
address_family_not_supported	EAFNOSUPPORT
address_in_use	EADDRINUSE
address_not_available	EADDRNOTAVAIL
already_connected	EISCONN
argument_list_too_long	E2BIG
argument_out_of_domain	EDOM
bad_address	EFAULT
bad_file_descriptor	EBADF
bad_message	EBADMSG
broken_pipe	EPIPE
connection_aborted	ECONNABORTED
connection_already_in_progress	EALREADY
connection_refused	ECONNREFUSED
connection_reset	ECONNRESET
cross_device_link	EXDEV
destination_address_required	EDESTADDRREQ
device_or_resource_busy	EBUSY
directory_not_empty	ENOTEMPTY
executable_format_error	ENOEXEC
file_exists	EEXIST
file_too_large	EFBIG
filename_too_long	ENAMETOOLONG
function_not_supported	ENOSYS
host_unreachable	EHOSTUNREACH
identifier_removed	EIDRM
illegal_byte_sequence	EILSEQ
inappropriate_io_control_operation	ENOTTY
interrupted	EINTR
invalid_argument	EINVAL
invalid_seek	ESPIPE
io_error	EIO
is_a_directory	EISDIR

(*continued*)

Table 8-2. (*continued*)

std::errc enum **Value**	**<cerrno> Macro**
message_size	EMSGSIZE
network_down	ENETDOWN
network_reset	ENETRESET
network_unreachable	ENETUNREACH
no_buffer_space	ENOBUFS
no_child_process	ECHILD
no_link	ENOLINK
no_lock_available	ENOLOCK
no_message	ENOMSG
no_message_available	ENODATA
no_protocol_option	ENOPROTOOPT
no_space_on_device	ENOSPC
no_stream_resources	ENOSR
no_such_device	ENODEV
no_such_device_or_address	ENXIO
no_such_file_or_directory	ENOENT
no_such_process	ESRCH
not_a_directory	ENOTDIR
not_a_socket	ENOTSOCK
not_a_stream	ENOSTR
not_connected	ENOTCONN
not_enough_memory	ENOMEM
not_supported	ENOTSUP
operation_canceled	ECANCELED
operation_in_progress	EINPROGRESS
operation_not_permitted	EPERM
operation_not_supported	EOPNOTSUPP
operation_would_block	EWOULDBLOCK
owner_dead	EOWNERDEAD
permission_denied	EACCES
protocol_error	EPROTO

(*continued*)

Table 8-2. (*continued*)

std::errc enum **Value**	**<cerrno> Macro**
protocol_not_supported	EPROTONOSUPPORT
read_only_file_system	EROFS
resource_deadlock_would_occur	EDEADLK
resource_unavailable_try_again	EAGAIN
result_out_of_range	ERANGE
state_not_recoverable	ENOTRECOVERABLE
stream_timeout	ETIME
text_file_busy	ETXTBSY
timed_out	ETIMEDOUT
too_many_files_open	EMFILE
too_many_files_open_in_system	ENFILE
too_many_links	EMLINK
too_many_symbolic_link_levels	ELOOP
value_too_large	EOVERFLOW
wrong_protocol_type	EPROTOTYPE

APPENDIX

Standard Library Headers

The C++ Standard Library consists of 88 header files, of which 6 are deprecated, and 26 are adapted from the C Standard Library. This appendix gives a brief description of each.

For each <*name*.h> header from the C Standard Library, there is a corresponding <c*name*> C++ Standard Library header (note the 'c' prefix). These C++ headers put all functionality provided by the C library in the `std` namespace. It is implementation-defined whether the types and functions still appear in the global namespace. The use of the original <name.h> headers is deprecated.

Headers are shown in the order in which they are presented in each chapter. Some are covered by multiple chapters. Functionality not discussed in this book is shown in italic.

Numerics and Math (Chapter 1)

Header	Contents
<cmath>	Math functions, such as `exp()`, `sqrt()`, `log()`, `abs()`, all trigonometric functions, special mathematical functions, and more.
<numeric>	The `gcd()` and `lcm()` functions (+ several algorithms: see Chapter 4).
<algorithm>	`min()`, `max()`, `minmax()`, and `clamp()` (+ many algorithms: Chapter 4).
<cstdint>	A set of type aliases for integral types with certain width requirements, e.g., `int32_t` and `int_fast64_t`.
<limits>	`numeric_limits`, offering properties—such as `min()`, `max()`, `lowest()`, `infinity()`, `quiet_NaN()`, etc.—for all built-in arithmetic types.
<climits>	*Macros for C-style limits of integral types, such as `INT_MAX`. Subsumed by <limits>.*
<cfloat>	*Macros to describe details of the floating-point types of your environment, e.g., `FLT_EPSILON`, `FLT_MAX`, etc. Subsumed by <limits>.*
<cfenv>	*Advanced access to the floating-point environment to configure floating-point exceptions, rounding, and other environment settings.*
<complex>	The `complex` class for working with complex numbers.

(*continued*)

P. Van Weert and M. Gregoire, *C++17 Standard Library Quick Reference*,
https://doi.org/10.1007/978-1-4842-4923-9

Header	Contents
<ccomplex>	*Simply includes <complex>. Deprecated since C++17.*
<ctgmath>	*Includes <cmath> and <ccomplex>. Deprecated since C++17.*
<ratio>	The ratio template, helper templates for performing arithmetic operations and comparisons on them, and a set of predefined ratios.
<random>	Pseudorandom number generators, random_device, and various random number distributions.
<valarray>	valarray functionality for working with arrays of numeric values.

General Utilities (Chapter 2)

Header	Contents
<utility>	pair, piecewise_construct, and *integer_sequence*. Functions make_pair(), swap(), exchange(), forward(), move(), move_if_noexcept(), as_const(), and declval().
<tuple>	tuple, helper classes tuple_size and tuple_element, and functions make_tuple(), forward_as_tuple(), tie(), tuple_cat(), get(), apply(), and make_from_tuple().
<cstddef>	The byte type.
<memory>	Smart pointers: unique_ptr, shared_ptr, and weak_ptr. Cast functions: static_pointer_cast(), dynamic_pointer_cast(), const_pointer_cast(), and reinterpret_pointer_cast(). The addressof() function. Also: allocators (Chapter 3) and algorithms for uninitialized memory (Chapter 4).
<new>	*Functions for managing dynamic storage: operators new, new[], delete, and delete[], get_ and set_new_handler(), and exceptions bad_alloc and bad_array_new_length.*
<functional>	Reference wrappers (created with ref() / cref()), predefined functors (function objects), std::bind(), not_fn(), function, and mem_fn(). The std::invoke() utility to invoke any callable.
<initializer_list>	The definition of initializer_list.
<optional> C++17	The optional class template, bad_optional_access exception, nullopt constant, and make_optional() function template.
<variant> C++17	The variant class template, the variant_size and variant_alternative templates, the bad_variant_access exception, monostate structure, variant_npos constant, and the functions visit(), holds_alternative(), get(), and get_if().

(*continued*)

Header	Contents
<any> C++17	The any class template, bad_any_cast exception, and any_cast() function template.
<chrono>	Time utilities: durations, time_points, and clocks (steady_clock, system_clock, and high_resolution_clock), and helper functions floor(), ceil(), round(), and abs().
<ctime>	C-style time and date utilities such as the tm struct, time(), localtime(), and strftime().
<typeinfo>	type_info, and the exceptions bad_cast and bad_typeid.
<typeindex>	type_index, a wrapper for type_info to be able to use it as a key in associative containers.
<type_traits>	Template-based type traits for compile-time manipulation and inspection of properties of types.

Containers (Chapter 3)

Header	Contents
<iterator>	Functions to perform operations on iterators: advance(), distance(), begin(), end(), prev(), and next(), and the iterator tags. See also Chapters 4 and 5.
<vector>	The vector class template and the vector<bool> specialization.
<deque>	The deque class template.
<array>	The array class template.
<list>	The list class template.
<forward_list>	The forward_list class template.
<bitset>	The bitset class template.
<queue>	The queue and priority_queue class templates.
<stack>	The stack class template.
<map>	The map and multimap class templates.
<set>	The set and multiset class templates.
<unordered_map>	The unordered_map and unordered_multimap class templates.
<unordered_set>	The unordered_set and unordered_multiset class templates.
<memory>	The default allocator type *and allocator-related utilities* allocator_traits, allocator_arg, and uses_allocator(). Also: smart pointers (Chapter 2) and algorithms for uninitialized memory (Chapter 4).

(*continued*)

Header	Contents
<memory_resource> C++17	The polymorphic allocators and memory resources.
<scoped_allocator> C++17	The scoped_allocator_adaptor class template.

Algorithms (Chapter 4)

Header	Contents
<iterator>	Input/output iterators, and the predefined iterator adaptors: reverse_iterator, move_iterator, and insert iterators. See also Chapters 3 and 5.
<algorithm>	80 different algorithms that operate on ranges.
<numeric>	Numerical algorithms: accumulate(), [transform_]reduce(), inner_product(), adjacent_difference(), partial_sum(), [transform_]inclusive_scan(), [transform_]exclusive_scan(), and iota().
<memory>	Algorithms for uninitialized memory: destroy[_n](), destroy_at(), and uninitialized_xxx(), with xxx equal to default_construct[_n], value_construct[_n], copy[_n], move[_n], and fill[_n]. Also: smart pointers (Chapter 2) and allocators (Chapter 3).
<execution> C++17	Predefined execution policies: seq, par, and par_unseq.

Input/Output (Chapter 5)

Header	Contents
<ios>	ios_base, basic_ios, and fpos, type aliases ios and wios, and types streamoff, streampos, wstreampos, and streamsize. Nonparametric I/O manipulators such as boolalpha, dec, scientific, etc.
<iomanip>	Parametric I/O manipulators, such as setbase(), setfill(), get_money(), put_time(), and more.
<ostream>	basic_ostream, and type aliases ostream and wostream. The endl, ends, and flush output manipulators.
<istream>	basic_istream and basic_iostream, and type aliases istream, wistream, iostream, and wiostream. The ws input manipulator.

(*continued*)

Header	Contents
<iostream>	cin/wcin, cout/wcout, cerr/wcerr, and clog/wclog. Includes <ios>, <streambuf>, <istream>, <ostream>, and <iosfwd>.
<sstream>	String streams: basic_istringstream, basic_ostringstream, basic_stringstream, basic_stringbuf, and related type aliases.
<fstream>	File streams: basic_ifstream, basic_ofstream, basic_fstream, and basic_filebuf, and related type aliases.
<streambuf>	basic_streambuf, and type aliases streambuf and wstreambuf.
<iosfwd>	Forward declarations for all stream I/O types.
<cstdio>	The C-style I/O library. Basic file utilities remove(), rename(), tmpfile(), tmpnam(), *plus fopen(), fclose(), etc.* Functions for formatted (printf(), scanf(), etc.) *and character-based I/O (getc(), putc(), etc.).* It is generally recommended to use C++ I/O streams.
<cinttypes>	*Macros to use with printf() and scanf() to handle the fixed-width integer types of <cstdint>. Subsumed by C++ I/O streams.*
<strstream>	*Deprecated.*
<iterator>	Stream iterators istream_iterator and ostream_iterator. See also Chapters 3 and 4.
<filesystem> C++17	Classes and functions to work with the file system.

Characters and Strings (Chapter 6)

Header	Contents
<string>	basic_string, and type aliases string, wstring, u16string, and u32string. Conversion functions such as stoi(), stof(), to_string(), etc.
<string_view> C++17	basic_string_view, and type aliases string_view, wstring_view, u16string_view, and u32string_view.
<cstring>	*Low-level memory functions: memcpy(), memmove(), memcmp(), memchr(), and memset(). A collection of C-style string functions, e.g., strcpy() and strcat(), and a definition for NULL and size_t.*
<cwchar>	*Functions to work with C-style wide character strings, such as fputws(), wprintf(), wcstof(), wcscat(), wmemset(), etc.*
<cctype>	Functions to classify and transform characters: isdigit(), isspace(), tolower(), toupper(), etc.

(*continued*)

Header	Contents
<cwctype>	Wide character versions of functions from <cctype>: iswdigit(), iswspace(), towlower(), towupper(), etc.
<codecvt>	Unicode character-encoding conversion facets: codecvt_utf8, codecvt_utf16, and codecvt_utf8_utf16. Deprecated since C++17.
<cuchar>	*Functions to convert between 16- or 32-bit character and multibyte sequences: c16rtomb(), c32rtomb(), mbrtoc16(), mbrtoc32().*
<locale>	The locale class, overloads of <cctype> functions accepting a given locale, facet functions use_facet() and has_facet(), and standard facet classes num_get, collate, money_put, codecvt, etc.
<clocale>	lconv and the setlocale() and localeconv() functions. setlocale() only changes the C locale.
<regex>	Everything related to regular expressions.
<charconv> C++17	The from_chars() and to_chars() functions, and chars_format class.

Concurrency (Chapter 7)

Header	Contents
<thread>	The thread class and the this_thread namespace.
<future>	future and shared_future, future_error, and providers promise, packaged_task, and async().
<mutex>	mutex, recursive_mutex, timed_mutex, recursive_timed_mutex, lock_guard, unique_lock, scoped_lock, and related types. Functions try_lock(), lock(), and call_once().
<shared_mutex>	shared_mutex, shared_timed_mutex, and shared_lock.
<condition_variable>	condition_variable and condition_variable_any, and the function notify_all_at_thread_exit().
<atomic>	Atomic types and fences.

Diagnostics (Chapter 8)

Header	Contents
<cassert>	The assert() macro.
<exception>	exception and bad_exception, exception pointers, nested exceptions, terminate and unexpected handlers.
<stdexcept>	Exception classes for reporting common errors: logic_error, runtime_error, and their generic subclasses.
<system_error>	The std::system_error exception used to report low-level errors, and the concepts of error codes, conditions, and categories.
<cerrno>	The errno expression, and default error condition values.

The C Standard Library

This section lists the remaining C headers that are not mentioned earlier.

Header	Contents
<ciso646>	*Only useful for C. Defines macros such as and, or, not, etc. In C++, those are reserved words, so this header is empty.*
<csetjmp>	*longjmp() and setjmp(). Do not use these in C++.*
<csignal>	*signal() and raise(). Do not use these in C++.*
<cstdalign>	*The __alignas_is_defined macro: always expands to 1 for C++. Deprecated since C++17.*
<cstdarg>	*The va_list type and functions va_start(), va_arg(), va_end(), and va_copy() to handle variable-length argument lists. In C++ it is recommended to use type-safe variadic templates instead.*
<cstdbool>	*The __bool_true_false_are_defined macro: expands to 1 for C++. Deprecated since C++17.*
<cstddef>	*Types ptrdiff_t, size_t, max_align_t, and nullptr_t. The macro offsetof() and the constant NULL.*
<cstdlib>	*String conversion functions: atof(), strtof(), etc.* *Multibyte character functions: mblen(), mbtowc(), and wctomb().* *Multibyte string conversion: mbstowcs() and wcstombs().* *Searching and sorting: bsearch() and qsort() (use <algorithm>).* *Random numbers: rand() and srand() (deprecated, use <random>).* *Memory management: calloc(), free(), malloc(), and realloc().* *Integer functions: abs(), div(), labs(), ldiv(), llabs(), and lldiv().* *Functions abort(), atexit(), at_quick_exit(), exit(), getenv(), quick_exit(), system(), and _Exit().*

Index

_1, _2, etc., 44
"C" locale, 200
<<, xxii, 157
>>, 157

A

abs(), 1
absolute(), 171
Absolute path, 163, 171
accumulate(), 132
acos(), 3
acosh(), 3
Active thread, 233
add_const, 67
add_cv, 67
add_lvalue_reference, 67
add_pointer, 67
addressof, 72
add_rvalue_reference, 67
add_volatile, 67
adjacent_difference(), 135
adjacent_find(), 118
adjustfield, 144
adopt_lock, 241, 242
advance(), 76
Aggregate, 132
<algorithm>, 115
Aliasing (shared_ptr), 41
alignment_of, 66
allocate_shared, 108
allocators, 108
all_of(), 117
any, 55
any_cast(), 56
any_of(), 117
app, 154
Appending, 79, 167
apply(), 35
Argument-dependent lookup (ADL), 32
Arithmetic type properties, 11
array, 84
ASCII, 189
as_const, 71
asctime(), 60
As-if rule, 249
asin(), 3
asinh(), 3
Assertions, 257
Associative containers
    ordered, 93
    unordered, 103
assoc_laguerre(), 7
assoc_legendre(), 7
async(), 235
Asynchronous programming, 235
    See also Futures
atan(), 3
atan2(), 3
atanh(), 3
ate, 154
<atomic>, 250
atomic_flag, 255
atomic_signal_fence(), 255
atomic_thread_fence(), 255
Atomic Variables, 250
    compare_exchange(), 252
    construction, 251
    exchange(), 252
    integral and pointer types, 251, 254
    lock_free(), 253
    non-member functions, 255

P. Van Weert and M. Gregoire, *C++17 Standard Library Quick Reference*,
https://doi.org/10.1007/978-1-4842-4923-9

Atomic Variables (*cont.*)
specializations, 251
store() and load(), 252
synchronization, 254
auto_format, 164
auto_ptr, 39
available, 178
Available disk space, 177

B

back_inserter(), 139
back_insert_iterator, 138
bad_alloc, 233, 248, 258
bad_any_cast, 56, 258
bad_array_new_length, 258
badbit, 147
bad_cast, 258
bad_exception, 258
bad_function_call, 45, 258
bad_optional_access, 48, 50, 258
bad_typeid, 258
bad_variant_access, 52, 258
bad_weak_ptr, 258
basefield, 144
basic_string, 189
basic_string_view, 195
begin(), 75
bernoulli_distribution, 19
Bessel functions, 6
beta(), 9
Bidirectional iterator, 73
bidirectional_iterator_tag, 74
binary, 154
binary_function, 42
binary_search(), 119
bind(), 44
bind1st(), 42
bind2nd(), 42
Binding function arguments, 44
binomial_distribution, 19
bit_and, 43
bit_not, 43
bit_or, 43
bitset, 89
bit_xor, 43
boolalpha, 144
bool_constant, 63
boyer_moore_horspool_searcher, 120
boyer_moore_searcher, 120
byte, 35

C

call_once(), 245
canonical(), 170
Capacity, 80, 178
CAS operations, 252
<cassert>, 257
cauchy_distribution, 19
cbegin(), 75
cbrt(), 2
<cctype>, 195
ceil(), 3
cend(), 75
cerr, 150
<cerrno>, 262, 264
C error numbers, 264
Character classes, 196, 207
Character classification, 196, 207
Character-encoding conversion, 197
Character encodings, 189, 197
<charconv>, 226, 229
chars_format, 229
char16_t, 189
char32_t, 189
chi_squared_distribution, 19
<chrono>, 56
chrono_literals, 57
cin, 152
clamp(), 10
classic(), 202
<clocale>, 213
C locales, 213
clock(), 60
Clocks, 59
CLOCKS_PER_SEC, 60
clock_t, 60
clog, 150
cmatch, 219
<cmath>, 1, 5
codecvt, 198
codecvt_byname, 199
codecvt_utf8, 198
codecvt_utf16, 198
codecvt_utf8_utf16, 198
collate, 208
common_type, 67
Compare-and-swap, 252
comp_ellint_1(), 7
comp_ellint_2(), 8
<complex>, 13
complex_literals, 13

Complex numbers, 13
Concatenation, 79, 167
conditional, 67
condition_variable, 246
condition_variable_any, 246
Condition Variables, 246
    exceptions, 248
    notification, 247
    synchronization, 249
    timeouts, 247
    waiting, 246
conjunction, 70
constexpr if, 68
Container adaptors, 91
Containers, 73
copy(), 123, 176
copy_backward(), 123
copyfmt_event, 158
copy_if(), 123
copy_n(), 123
copy_options, 177
copysign(), 4
copy_symlink(), 169
copy_symlinks, 177
cos(), 3
cosh(), 3
count(), 117
count_if(), 117
cout, 150
crbegin(), 75
create_directories(), 176
create_directory(), 176
create_directory_symlink(), 169
create_hard_link(), 169
create_hard_links, 177
create_symlink(), 169
create_symlinks, 177
cref(), 43, 46
crend(), 75
Critical section, 249 *See also* Mutexes
<cstdint>, 10
<cstdio>, 180, 181
C-style date and time utilities, 60
csub_match, 219
ctime(), 60
ctype, 207
Cumulative sum, 134
Currency symbol, 205
current_exception(), 259
current_path(), 170
Current thread, 233
<cwctype>, 195
cyl_bessel_i(), 6
cyl_bessel_j(), 6
cyl_bessel_k(), 6
cyl_neuman(), 6

D

data() (non-member function), 89
Data race, 137, 238, 250
Date formatting, 206
Date utilities, 60
Deadlock, 137, 233, 240, 244
dec, 144
decay, 67
Decimal separator, 203
declval(), 70
defaultfloat, 145
default_random_engine, 17
default_searcher, 120
defer_lock, 242
deque, 83
destroy(), 136
destroy_at(), 136
destroy_n(), 136
Difference, 130
difftime(), 60
Digit grouping, 203
Directories, 162, 178
directories_only, 177
directory_entry, 178
directory_iterator, 178
Directory listing, 178
directory_options, 179
Directory separator, 162
discard_block_engine, 16
discrete_distribution, 20
disjunction, 70
Disk space, 177
distance(), 76
Distribution, *see* Random number distributions
div(), 2
divides, 43
domain_error, 258
Dot product, 133, 160
Double-checked locking, 245
Double-ended queue, 83
Doubly linked list, 84
duration, 57
duration_cast(), 57

E

ECMAScript grammar, *see* Regular expressions
ellint_1(), 7
ellint_2(), 7
Emplacement, 79, 95–96
empty() (non-member function), 89
enable_if, 68
enable_shared_from_this, 41
end(), 75
endl, 150
ends, 150
eofbit, 147
Epoch, 58
Epsilon, 12
equal(), 122
equal_range(), 119
equal_to, 43
equivalent(), 172
erase_event, 158
erf(), erfc(), 8
errc, 229, 262
errno, 264
error_category, 262
error_code, 163, 229, 262
error_condition, 262
Error functions, 8
Exception pointers, 259
exception_ptr, 259
Exceptions, 148, 163
Exceptions class hierarchy, 258
exchange(), 32
exclusive_scan(), 134
execution namespace, 136
exists(), 173
exp(), 2
exp2(), 2
expint(), 8
expm1(), 2
exponential_distribution, 20
Extension, *see* File extension
extent, 67
extreme_value_distribution, 20

F

fabs(), fabsf(), fabsl(), 1
Facets, 202 *See also* Localization
failbit, 147
failure, 258
False sharing, 248
false_type, 63, 69
fdim(), 2
Fences, 249, 254, 255
File extension, 166, 168
File links, 168
File permissions, 174
Files, 162
file_size(), 177
file_status, 172
File streams, 155
<filesystem>, 162
filesystem_error, 163
File types, 173
fill(), 122
Fill character, 145, 149
fill_n(), 122
find(), 118
find_end(), 120
find_first_of(), 118
find_if(), 117
find_if_not(), 117
Fire-and-forget, 232, 236
First-in first-out (FIFO), 91
fisher_f_distribution, 19
fixed, 144
Fixed-width integer types, 10
floatfield, 144
Floating-point numbers
  Epsilon, 12
  Infinity, 12
  NaN, 2, 4, 13
floor(), 3
flush, 150
fma(), 2
fmax(), 2
fmin(), 2
fmod(), 1
fmtflags, 144
Fold, 132
follow_directory_symlink, 179
for_each(), 115
for_each_n(), 116
Formatting, *See also* to_string(), to_chars(), printf(), Regular expressions, and Stream I/O
  date, 61, 206
  monetary, 146
  numerical, 144
  time, 61, 206

forward(), 31
forward_as_tuple(), 35
Forwarding reference, 31
Forward iterator, 73
forward_iterator_tag, 75
forward_list, 84
fpclassify(), 4
FP_INFINITE, 4
FP_NAN, 4
FP_NORMAL, 4
fpos, 142
fprintf(), 181
FP_SUBNORMAL, 4
FP_ZERO, 4
free, 178
Free space, 177
frexp(), 3
from_chars(), 229
front_inserter(), 139
front_insert_iterator, 138
fscanf(), 185
<fstream>, 155
function, 45
<functional>, 42, 71
Function object, 42
    for class members, 46
Functor, 42
<future>, 234
future_errc, 238
future_error, 237, 258
Futures, 234
    exceptions, 237
    providers, 234, 235
        async(), 235
        packaged tasks, 236
        promises, 237
    shared state, 234
    synchronization, 250

G

gamma_distribution, 20
Gamma functions, 8
gcd(), 4
generate(), 122
generate_canonical, 19
generate_n(), 122
Generic function wrappers, 45
generic_format, 162
geometric_distribution, 19
get_default_resource(), 109
get_if() (variant), 53
getline(), 151, 153
get_money(), 146
get_terminate(), 266
get_time(), 146
get() (tuple), 34
get() (variant), 52
global(), 202
gmtime(), 60
goodbit, 147
Grammar
    printf(), 181
    regular expressions, 214
    scanf(), 185
    time and date formatting, 61
greater, 43
greater_equal, 43
gslice, 25

H

hard_link_count(), 169
hardware_concurrency(), 233
hardware_constructive_
        interference_size, 248
hardware_destructive_
        interference_size, 248
has_facet(), 202
hash, 105
Hash functions, 104
Hash map, 104
has_unique_object_respresentations, 66
Header
    <algorithm>, 115
    <any>, 55
    <array>, 84
    <atomic>, 250
    <bitset>, 89
    <cassert>, 257
    <cctype>, 195
    <cerrno>, 262, 264
    <charconv>, 226, 229
    <chrono>, 56
    <clocale>, 213
    <cmath>, 1, 5
    <codecvt>, 197
    <complex>, 13
    <condition_variable>, 246
    <cstdint>, 10
    <cstdio>, 180, 181
    <ctime>, 60

Header (*cont.*)
<cwctype>, 195
<deque>, 83
<exception>, 258
<filesystem>, 162
<forward_list>, 84
<fstream>, 155
<functional>, 42, 71
<future>, 234
<initializer_list>, 47
<iomanip>, 145
<ios>, 142, 143
<iosfwd>, 142
<iostream>, 142, 150
<istream>, 151
<iterator>, 73, 113, 138
<limits>, 11
<list>, 84
<locale>, 200
<map>, 94, 98
<memory>, 36, 108
<memory_resource>, 108
<mutex>, 238
<numeric>, 132
<optional>, 48
<ostream>, 149
<queue>, 91
<random>, 15
<ratio>, 14
<regex>, 214
<scoped_allocator>, 111
<set>, 98
<shared_mutex>, 240
<sstream>, 153
<stack>, 92
<stdexcept>, 258
<streambuf>, 161
<string>, 189, 227
<string_view>, 194
<system_error>, 262
<thread>, 231
<tuple>, 34
<typeindex>, 62
<typeinfo>, 62
<type_traits>, 63
<unordered_map>, 103
<unordered_set>, 103
<utility>, 29, 33
<valarray>, 23
variant, 50
<vector>, 76
Heaps, 131
hermite(), 7
hex, 144
hexfloat, 145
high_resolution_clock, 59
Hints, 97
holds_alternative(), 52
hypot(), 2

I, J

ifstream, 155
ilogb(), 3
imbue_event, 158
in, 154
includes(), 129
inclusive_scan(), 134
independent_bits_engine, 17
indirect_array, 27
Infinity, 12
<initializer_list>, 47
Initializer-list constructors, 47
inner_product(), 133
in_place_index, in_place_index_t, 51
inplace_merge(), 129
in_place_type, in_place_type_t, 51–52, 55
Input iterators, 113
input_iterator_tag, 113
Input streams, *see* Stream I/O
inserter(), 139
insert_iterator, 139
integral_constant, 63
Integrals, 7, 8
int_fastX_t, 10
int_leastX_t, 10
internal, 144
Internationalization, *see* Localization
Intersection, 130
intmax_t, 11
intptr_t, 11
intX_t, 10
invalid_argument, 258
invoke, 71
invoke_result, 69
I/O, *see* Stream I/O
<iomanip>, 145
I/O Manipulator, *see* Stream I/O
I18n, *see* Localization
<ios>, 142, 143
ios_base, 143
<iosfwd>, 142

iostate, 147
iostream, 142, 150, 153
iota(), 123
is_abstract, 65
is_aggregate, 66
is_alnum(), 196, 207
is_alpha(), 196, 207
is_arithmetic, 64
is_array, 64
is_assignable, 65
is_base_of, 66
is_blank(), 196, 215
is_block_file(), 173
is_character_file(), 175
is_class, 64
is_cntrl(), 196, 215
is_const, 65
is_constructible, 65
is_convertible, 66
is_copy_assignable, 65
is_copy_constructible, 65
is_default_constructible, 65
is_destructible, 65
is_digit(), 196, 207
is_directory(), 173
is_empty, 66
is_enum, 64
is_error_code_enum, 263
is_error_condition_enum, 264
is_execution_policy_v, 137
is_fifo(), 173
is_final, 65
isfinite(), 4
is_floating_point, 64
is_function, 64
is_fundamental, 64
is_graph(), 196, 207
isgreater(), 4
isgreaterequal(), 4
is_heap(), 131
is_heap_until(), 132
isinf(), 4
is_integral, 64
is_invocable, 69
is_invocable_r, 69
isless(), 4
islessequal(), 4
islessgreater(), 4
is_literal_type, 66
is_lower(), 196, 208
is_lvalue_reference, 64
is_member_function_pointer, 64
is_member_object_pointer, 64
is_member_pointer, 64
is_move_assignable, 65
is_move_constructible, 65
isnan(), 4
isnormal(), 4
is_null_pointer, 64
is_object, 64
is_other(), 173
is_partitioned(), 126
is_permutation(), 130
is_pod, 66
is_pointer, 64
is_polymorphic, 65
is_print(), 196, 208
is_punct(), 196, 207
is_reference, 64
is_regular_file(), 173
is_rvalue_reference, 64
is_same, 66
is_scalar, 64
is_signed, 65
is_socket(), 173
is_sorted(), 127
is_sorted_until(), 127
is_space(), 196, 208
is_standard_layout, 66
is_swappable, is_swappable_with, 65
is_symlink(), 169, 173
<istream>, 151
istream_iterator, 160
istringstream, 153
is_trivial, 66
is_trivially_copyable, 65, 68
is_union, 64
isunordered(), 4
is_unsigned, 65
is_upper(), 196, 208
is_void, 64
is_volatile, 65
is_walnum(), 196
is_walpha(), 196
is_wblank(), 196
is_wcntrl(), 196
is_wdigit(), 196
is_wgraph(), 196
is_wlower(), 196
is_wprint(), 196
is_wpunct(), 196
is_wspace(), 196

is_wupper(), 196
is_wxdigit(), 196
is_xdigit(), 196, 215
iterator, 73, 113
Iterator adaptors, 138
Iterators
    bidirectional, 73
    categories, 73, 113
    forward, 73
    input, 113
    output, 113
    random access, 73
    reverse, 74, 75
    stream iterators, 160
Iterator tags, 74
iterator_traits, 75
iter_swap(), 124

K

knuth_b, 17

L

labs(), 1
laguerre(), 7
Last-in first-out (LIFO), 92
Launch policy, 236
Lazy initialization, 245
LC_ALL, 213
LC_COLLATE, 213
lcm(), 4
LC_MONETARY, 213
LC_NUMERIC, 213
lconv, 213
LC_TIME, 213
LC_CTYPE, 213
L1 data cache line size, 248
ldexp(), 4
ldiv(), 2
left, 144
Left fold, 132
legendre(), 7
length_error, 258
less, 43
less_equal, 43
lexicographical_compare(), 127
lgamma(), 8
<limits>, 11
linear_congruential_engine, 16
Line-by-line input, 153
list, 84
List-specific algorithms, 85
llabs(), 1
lldiv(), 2
llrint(), 3
llround(), 3
localeconv(), 213
Locale names, 200
Localization, 200
    C locales, 213
    combining facets, 210
    custom facets, 211
    global locale, 201
    locale facet categories, 203
    locale facets, 202
    locale names, 200
    standard facets, 203
        character classification and transformation, 207
        character-encoding conversions, 208
        formatting and parsing
            monetary values, 205
            numeric values, 204
            time and dates, 206
    message retrieval, 209
    monetary punctuation, 204
    numeric punctuation, 203
    string ordering and hashing, 208
    std::locale, 200
localtime(), 60
lock(), 244
lock_guard, 241, 276
Lock-free data structures, 252
Locks, *see* Mutexes
log(), 2
log2(), 2
log10(), 2
logb(), 3
log1p(), 2
logical_and, 43
logical_not, 43
logical_or, 43
logic_error, 258
lognormal_distribution, 19
lower_bound(), 119
lrint(), 3
lround(), 3

M

Magic statics, 245
main(), xxii
make_error_code(), 263
make_error_condition(), 264
make_exception_ptr(), 259
make_from_tuple, 35
make_heap(), 131
make_move_iterator(), 138
make_optional(), 49
make_pair(), 33
make_ready_at_thread_exit(), 237
make_reverse_iterator(), 138
make_shared(), 40
make_signed, 67
make_tuple(), 34
make_unique(), 37–38
make_unsigned, 67
Manipulator, *see* Stream I/O
map, 94
mask_array, 26
match_flag_type, 218, 224
match_results, 219, 220
Mathematical functions
  classification, 4
  common functions, 1
  comparison, 4
  error functions, 8
  error handling, 5
  exponential functions, 2
  floating-point manipulation, 3
  gamma functions, 8
  hyperbolic functions, 3
  logarithmic functions, 2
  power functions, 2
  rounding, 3
  special functions, 5
  trigonometric functions, 3
MATH_ERREXCEPT, 5
math_errhandling, 5
MATH_ERRNO, 4
max(), 9
max_element(), 118
Maximum representable number, 11
mbstate_t, 197
Member function object, 46
memcpy(), 68
mem_fn(), 46
mem_fun(), 42
mem_fun_ref(), 42
<memory>, 36, 108
Memory model, 249
Memory pool, 110
memory_order, 254
memory_resource, 108
merge(), 129
Merging, *See* Appending, Concatenation
Merging (associative containers), 100
mersenne_twister_engine, 16
messages, 209
min(), 9
min_element(), 118
Minimum representable number, 11
minmax(), 10
minmax_element(), 118
minstd_rand, 17
minstd_rand0, 17
minus, 43
mismatch(), 122
mktime(), 60
modf(), 3
modulus, 43
Monetary formatting, 146
money_get, 205
moneypunct, 204, 211
moneypunct_byname, 212
money_put, 205
monostate, 51
monotonic_buffer_resource, 110
move(), 29, 123
move_backward(), 123
move_if_noexcept(), 30
move_iterator, 138
Move semantics, 29
mt19937, 17
mt19937_64, 17
multimap, 98
multiplies, 43
multiset, 98
mutex, 238, 239
Mutexes, 238
  critical section, 249
  exceptions, 244
  lock types
    lock_guard, 241
    scoped_lock, 239, 241
    shared_lock, 243
    unique_lock, 242
  locking, 238, 241

Mutexes (*cont.*)
locking multiple mutexes, 244
native_handle(), 240
RAII, 238, 241
readers-writers, 240
recursion, 240
reentry, 240
sharing ownership, 240
synchronization, 249
timeouts, 240

N

NaN, 2, 4, 13
nan(), nanf(), nanl(), 2
native_format, 164
NDEBUG, 257
nearbyint(), 3
negate, 43
negation, 70
negative_binomial_distribution, 19
nested_exception, 260
Neutral locale, 200
new_delete_resource(), 109
next(), 76
nextafter(), 4
next_permutation(), 131
nexttoward(), 4
Nodes (associative containers), 100
none_of(), 117
normal_distribution, 19
not1(), 45
not2(), 45
not_equal_to, 43
not_fn(), 45
notify_all_at_thread_exit(), 247
npos, 190
nth_element(), 18
null_memory_resource(), 109
nullopt, nullopt_t, 48
<numeric>, 132
Numeric conversions, 226
numeric_limits, 11
Numerical formatting, 145
num_get, 204
numpunct, 203, 212
numpunct_byname, 212
num_put, 204

O

oct, 144
ofstream, 155
once_flag, 245
openmode, 154, 155
optional, 48
Ordered associative containers, 93
<ostream>, 149
ostream_iterator, 160
ostringstream, 154
out, 154
out_of_range, 258
Output iterators, 113
output_iterator_tag, 113
Output streams, *see* Stream I/O
overflow_error, 258
overwrite_existing, 177
owner_less, 41

P

packaged_task, 236
pair, 33
par, 137
Parallel algorithms, 136
parallel_policy, 137
parallel_unsequenced_policy, 137
Parsing, *see* stoi(), stof(), from_chars(), scanf(), Regular expressions, and Stream I/O
Parsing floating-point numbers, 228
Parsing integers, 227
partial_sort(), 127
partial_sort_copy(), 127
partial_sum(), 134
partition(), 126
partition_copy(), 126
partition_point(), 127
par_unseq, 137, 138
path, 164
Pathnames, 162, 166
Perfect forwarding, 31
Permissions, 174
permissions(), 175
perms, 174
Permutations, 130
perror(), 265
Person class, xxiii
Piecewise construction, 34, 96

piecewise_constant_distribution, 21
piecewise_linear_distribution, 22
Placeholders, 44
plus, 43
pmr (namespace), 108, 109
poisson_distribution, 20
polymorphic_allocator, 108
Polynomials, 7
pool_options, 111
pop_heap(), 131
POSIX error codes, 264, 267
pow(), 2
Predefined functors, 43
preferred_separator, 162
Prefix sum, 134
prev(), 76
prev_permutation(), 13
printf(), 181
    conversion specifiers, 182
    flags, 184
    formatting syntax, 183
    length modifiers, 185
priority_queue, 91
promise, 237
proximate(), 171
ptr_fun(), 42
push_heap(), 131
put_money(), 146
put_time(), 146

Q

queue, 91
quoted(), 146

R

RAII, 36, 241
rand(), 15
<random>, 15
Random access iterator, 74
Random number distributions, 18
    Bernoulli, 19
    Normal, 19
    Poisson, 20
    Sampling
        Discrete, 20
        Piecewise constant, 21
        Piecewise linear, 22
    Uniform, 19
Random number generators, 15
    Non-deterministic, 18
    Pseudorandom number engines, 16
        Engine adaptors, 16
        Predefined engines, 17
Random numbers, 15
    Seeding, 18
random_access_iterator_tag, 75
random_device, 18
range_error, 258
rank, 66
ranlux24, 17
ranlux24_base, 17
ranlux48, 17
ranlux48_base, 17
<ratio>, 14
ratio_add, 14
ratio_divide, 14
ratio_equal, 14
ratio_multiply, 14
Rational numbers, 14
ratio_subtract, 14
rbegin(), 75
Readers-writers locks, *see* Mutexes
read_symlink(), 169
recursive, 177
recursive_directory_entry, 179
recursive_mutex, 239
recursive_timed_mutex, 239
reduce(), 132
ref(), 43, 46
Reference wrappers, 43, 46
reference_wrapper, 43, 46
regex, 214, 216
regex_error, 216, 219, 223
regex_iterator, 221
regex_match(), 218
regex_replace(), 223
regex_search(), 218
regex_token_iterator, 222
Regular expressions, 214
    grammar, 214
        assertions, 214, 225
        atoms, 214, 224
        back reference, 214
        character classes, 215
        disjunction, 214
        greediness, 216
        lookahead, 226
        quantifiers, 226

Regular expressions (*cont.*)
grammar options, 216
matching and searching patterns, 218
match iterators, 221
match results, 219
raw string literals, 214
replacing patterns, 223
std::regex, 216
Relational operators, 36
relative(), 171
Relative path, 163, 171
release() (unique_ptr), 38
rel_ops, 36
remainder(), 1
remove(), 176
remove() (algorithm), 124, 275
remove() (file), 176, 180
remove_all(), 176
remove_const, 67
remove_copy(), 125
remove_copy_if(), 125
remove_cv, 67
Remove-erase idiom, 81, 124
remove_if(), 124
remove_lvalue_reference, 67
remove_pointer, 67
remove_rvalue_reference, 67
remove_volatile, 67
remquo(), 1
rename(), 176, 180
rend(), 75
replace(), 125
replace_copy(), 125
replace_copy_if(), 125
replace_if(), 125
reset() (shared_ptr), 40
reset() (unique_ptr), 38
resetiosflags(), 145
resize_file(), 177
Resource Acquisition Is Initialization, *see* RAII
result_of, 70
rethrow_exception(), 259
rethrow_if_nested(), 261
reverse(), 126
reverse_copy(), 126
Reverse iterator, 74, 75
reverse_iterator, 138
riemann_zeta(), 9
Right fold, 132
right, 144
rint(), 3
rotate(), 126
rotate_copy(), 126
round(), 3
runtime_error, 258
Runtime type identification, 62

S

sample(), 128
Scalar product, 133
scalbln(), scalbn(), 4
Scan, 134
scanf(), 185
conversion specifiers, 186
formatting syntax, 186
length modifiers, 187
scientific, 144
<scoped_allocator>, 111
scoped_allocator_adaptor, 111
scoped_lock, 239, 241
search(), 120
search_n(), 120
seekdir, 149
Selection algorithm, 128
seq, 137
Sequence comparison, 121
sequenced_policy, 137
Sequential containers, 76
set, 98
setbase(), 145
set_default_resource(), 109
set_difference(), 130
set_exception_at_thread_exit(), 237
setfill(), 145
set_intersection(), 130
setiosflags(), 145
setlocale(), 213
setprecision(), 145
set_symmetric_difference(), 130
set_terminate(), 266
set_union(), 130
set_value_at_thread_exit(), 237
setw(), 146
SFINAE, 68
shared_from_this(), 41
shared_future, 234
shared_lock, 243
shared_mutex, 239, 276
shared_ptr, 39
shared_timed_mutex, 239

showbase, 144
showpoint, 144
showpos, 144
shuffle(), 128
shuffle_order_engine, 17
signbit(), 4
sin(), 3
Singleton, 245
sinh(), 3
SI ratios, 4
size() (non-member function), 89
skip_existing, 177
skip_permission_denied, 179
skip_symlinks, 177
skipws, 144
sleep_for(), 233
Sleeping, 233
sleep_until(), 233
slice, 24
Smart pointers, 36 *See also* RAII.
smatch, 219
sort(), 128
sort_heap(), 131
space(), 178
space_info, 178
sph_bessel(), 6
sph_legendre(), 7
sph_neuman(), 6
Splicing, 85
Splitting strings, *see* regex_token_iterator
sprintf(), 181
Spurious wakeups, 246
sqrt(), 2
sscanf(), 185
<sstream>, 153
ssub_match, 219
stable_partition(), 126
stable_sort(), 127
stack, 92
Standard Template Library, xix
status(), 172
status_known(), 173
std, xxi
stderr, 150
<stdexcept>, 258
stdin, 152
stdout, 150
steady_clock, 59
STL, xix
stod(), 228
stof(), 228
stoi(), 227
stol(), 227
stold(), 228
stoll(), 227
stoul(), 227
stoull(), 227
<streambuf>, 161
Stream Buffers, 161
Stream I/O
  class hierarchy, 141
  default initialization, 145, 148
  error handling, 148
  file streams, 155
  formatting flags, 144
  helper types, 142
  input streams, 151
  I/O manipulators, 145, 150
  open modes, 155
  output streams, 149
  standard input streams, 152
  standard output streams, 150
    redirect, 161
  state bits, 148
  stream iterators, 160
  string streams, 153
  thread safety, 151
streamoff, 142
streamsize, 142
strerror(), 265
strftime(), 60
<string>, 189, 227
string_literals, 192
Strings, 189
  comparing, 193
  constructing, 192
  formatting (*see* Formatting)
  length, 192
  modifying, 191
  npos, 190
  parsing (*see* Parsing)
  searching, 190
  string literal operator, 192
  substrings, 193
  types, 189
String streams, 153
string_view, 194
string_view_literals, 195
student_t_distribution, 19
sub_match, 219
Subsequence search, 120

Substrings, 193
subtract_with_carry_engine, 16
Summation, 132
sv (""sv), 195
swap(), 32
swap_ranges(), 123
Symbolic link, 168
symlink_status(), 172
Symmetric difference, 130
Synchronization, 249 *See also* Memory model
synchronized_pool_resource, 110
sync_with_stdio(), 151, 152
system_clock, 59
system_error, 163, 229, 233, 237, 238, 244, 248, 258

T

tan(), 3
tanh(), 3
terminate(), 266
tgamma(), 8
Thousands separator, 203
<thread>, 231
Threads, 231
  exceptions, 233
  fire-and-forget, 232, 236
  identifiers, 232
  joining, 232
  launching, 231
  sleeping, 233
  synchronizing, 250
  yielding, 233
throw_with_nested(), 261
tie(), 34
time(), 60
timed_mutex, 239
Time formatting, 146
time_get, 206
time_point, 58
time_point_cast(), 58
time_put, 206
time_t, 60
Time utilities, 56
tm, 60
tmpfile(), 180
tmpnam(), 180
to_chars(), 229
Tokenizing, *see* regex_token_iterator
tolower(), 196, 208
to_string(), 90
toupper(), 196, 208
Torn reads and writes, 252 *See also* Atomic variables
towlower(), 196
to_wstring(), 228
towupper(), 196
transform(), 116
transform_exclusive_scan(), 135
transform_inclusive_scan(), 135
transform_reduce(), 132, 133
Transparent operator functors, 44, 99
True sharing, 248
true_type, 63, 69
trunc, 154
trunc(), 3
try_lock(), 244
try_to_lock, 242
<tuple>, 34
tuple_element, 35
tuple_size, 35
Type classification, 64
typeid(), 62
<typeindex>, 62
<typeinfo>, 62
Type properties, 65
Type property queries, 66
Type relationships, 66
Type traits, 63
<type_traits>, 63
Type transformations, 67

U

u16string, 189
u16string_view, 195
u32string, 189
u32string_view, 195
u8path(), 164
(u)int_fastX_t, 10
(u)int_leastX_t, 10
(u)intmax_t, 11
uintptr_t, 11
(u)intX_t, 10
unary_function, 42
uncaught_exceptions(), 265, 266
underflow_error, 258
underlying_type, 67
Unicode, 192, 197
uniform_int_distribution, 19
uniform_real_distribution, 19

uninitialized_copy(), 136
uninitialized_copy_n(), 136
uninitialized_default_construct(), 135
uninitialized_fill(), 136
uninitialized_fill_n(), 136
uninitialized_move(), 136
uninitialized_move_n(), 136
uninitialized_value_construct(), 135
Union, 130
unique(), 125
unique_copy(), 125
unique_lock, 242
unique_ptr, 36
unitbuf, 144
Universal reference, 31
Unordered associative containers, 103
unordered_map, 103
unordered_multimap, 103
unordered_multiset, 103
unordered_set, 103
unsynchronized_pool_resource, 110
update_existing, 177
upper_bound(), 119
uppercase, 144
Upstream memory resource, 110
use_facet(), 202
UTF-8, UTF-16, UTF-32, 189, 197
<utility>, 29, 33, 272

V

<valarray>, 23
variant, 50
variant_alternative,
variant_alternative_t, 54
variant_size, variant_size_v, 54
vector, 76
vector<bool>, 82
visit(), 53
void_t, 69

W, X

wbuffer_convert, 199
wcerr, 150
wchar_t, 189
wcin, 152
wclog, 150
wcmatch, 219
wcout, 150
wcsub_match, 219
weak_from_this(), 41
weakly_canonical(), 170
weak_ptr, 41
Weak reference, 41
weibull_distribution, 20
Working directory, 170
ws, 152
wsmatch, 219
wssub_match, 219
wstring, 189
wstring_convert, 198
wstring_view, 195

Y

yield(), 233

Z

Zeta functions, 9