THE EXPERT'S VOICE® IN C++

C++ Standard Library Quick Reference

Peter Van Weert
Marc Gregoire

Apress®

C++ Standard Library Quick Reference

Peter Van Weert
Marc Gregoire

Apress®

C++ Standard Library Quick Reference

Peter Van Weert
Kessel-Lo, Belgium

Marc Gregoire
Meldert, Belgium

ISBN-13 (pbk): 978-1-4842-1875-4
ISBN-13 (electronic): 978-1-4842-1876-1
DOI 10.1007/978-1-4842-1876-1

Library of Congress Control Number: 2016941348

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Technical Reviewer: Bart Vandewoestyne
Editorial Board: Steve Anglin, Pramila Balan, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing
Coordinating Editor: Mark Powers
Copy Editor: Tiffany Taylor
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

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 rights@apress.com, or visit www.apress.com.

Apress and friends of ED books 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 Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com/9781484218754. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

To my parents and my brother and his wife.
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 at a Glance

Contents

About the Authors

Peter Van Weert is a Belgian software engineer whose main interest and expertise are programming languages, algorithms, and data structures.

He received his master's 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 on the design and efficient compilation of rule-based programming languages at the research group for declarative programming languages and artificial intelligence of the same university. During his doctoral studies, he was a teaching assistant for object-oriented programming (Java), software analysis and design, and declarative programming.

After graduating, Peter joined Nikon Metrology to work on large-scale, industrial application software in the area of 3D laser scanning and point cloud inspection. At Nikon, he has mastered C++ and refactoring and debugging of very large code bases and has gained further proficiency in all aspects of the software development process, including the analysis of functional and technical requirements, and agile and scrum-based project and team management.

In his spare time, he has co-authored two award-winning Windows 8 apps, and he is a regular speaker at and board member of the Belgian C++ Users Group.

Marc Gregoire is a software engineer from Belgium. He graduated from the University of Leuven, Belgium, with a degree in "Burgerlijk ingenieur in de computer wetenschappen" (equivalent to a master's of science in engineering in computer science). The year after, he received the cum laude degree of master's in artificial intelligence 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 USA to EMEA and Asia. Now, Marc is working for Nikon Metrology on industrial 3D laser scanning software.

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# and uses PHP for creating web pages.

Since April 2007, he has received the yearly 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++*, and a member on the CodeGuru forum (as Marc G). He maintains a blog at www.nuonsoft.com/blog/.

About the Technical Reviewer

Bart Vandewoestyne is an enthusiastic, solo-parenting software engineer living in Belgium. After obtaining his master's degree from the Computer Science department at the University of Leuven, he worked as a researcher in the numerical analysis and applied mathematics section of that same university. He successfully completed his PhD in 2008. Three years of postdoctoral work later, Bart left the academic world for industry. He now works as a senior development software engineer for Esterline Belgium, where he develops and maintains software for professional flight-simulator alignment.

Bart enjoys reading about and exploring all aspects of C++ software development. In his spare time, and when he's away from his keyboard, he enjoys running, swimming, paragliding, and spending quality time with his now 6-year-old son Jenne. He wants the world to know how much he cares about Jenne, and he hopes that his child will also transform his passion into his profession.

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 20 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.

The C++ Standard Library today is much more than the STL containers and algorithms. For decades, it has featured STL-like string classes, extensive localization facilities, and a stream-based I/O library, as well as all headers of the C Standard Library. In recent years, 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. Many of these libraries 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 few years. The next version of the Standard, tentatively called C++17, 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 over 750 pages in length, and those on the

C Standard Library encompass another 250 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++14 and C11 Standard Libraries, some in more detail than others, and 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 file and stream-based I/O
- 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 C Standard Library headers 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 the C headers if there are no more modern, C++-style alternatives provided by the C++ Standard Library.
- 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++14 version of the Standard. We occasionaly 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.

Unless otherwise noted, 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 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 website (`www.apress.com/9781484218754`).

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

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 operator, even though these facilities of the C++ I/O library are only discussed in detail in Chapter 5. The stream operator can be used to output virtually all fundamental C++ types, and multiple values can be written on a single line. The 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(const std::string& first,
                   const std::string& last = "", bool isVIP = false)
      : m_first(first), m_last(last), m_isVIP(isVIP) {}

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

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

   bool IsVIP() const { return m_isVIP; }
```

```
private:
   friend bool operator<(const Person&, const Person&);
   std::string m_first;
   std::string 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();
}

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

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)	Returns the absolute value of x. <cstdlib> defines abs(), labs(), and llabs() for int types; that abs() has return type int, which is different from the abs() in <cmath> (double).
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. E.g.: 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 the nearest).

(*continued*)

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-1876-1_1) contains supplementary material, which is available to authorized users.

P. Van Weert and M. Gregoire, *C++ Standard Library Quick Reference*,
DOI 10.1007/978-1-4842-1876-1_1

Function	Description
`fma(x, y, z)`	Computes $(x*y)+z$ in an accurate (better precision and rounding properties than a naive 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`.
`fdim(x, y)`	Returns the positive difference, i.e. $\begin{cases} x-y \; if \; x>y \\ +0 \; if \; x \le y \end{cases}$.

`nan(string)`
`nanf(string)`
`nanl(string)`
	Returns a quiet (non-signaling) NaN (not a number) of type `double`, `float`, respectively `long double`, 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)`	$2x$	`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)`	$\sqrt[3]{x}$

Trigonometric and Hyperbolic Functions

All basic trigonometric (`sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`) and hyperbolic functions (`sinh()`, `cosh()`, `tanh()`, `asinh()`, `acosh()`, `atanh()`) are provided. The lesser-known trigonometric function `atan2()` is provided as well. It is used to compute the angle between a vector (`x`,`y`) and the positive X axis, with `atan2(y,x)` 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 `0`). Essentially, by dividing `y` by `x` in `atan(y/x)`, you lose information regarding the sign of `x` and `y`.

Error and Gamma Functions

Function	Formula	Function	Formula
`erf(x)`	$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$	`tgamma(x)`	$\Gamma(x)=\int_0^\infty t^{x-1}e^{-t}dt$
`erfc(x)`	$\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}dt = 1-\mathrm{erf}(x)$	`lgamma(x)`	$\ln(\lvert\Gamma(x)\rvert)$

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`, whereas `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`, whereas `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 a fractional part. The latter is returned and 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`, that is: $\log_{radix}\lvert x\rvert$, with *radix* the base used to represent floating-point values (`2` for all standard numerical types; hence the name: *binary logarithm*).

(*continued*)

Function	Description
`ilogb(x)`	Same as `logb(x)`, but the result is truncated to a signed `int`.
`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 NaN.
`isinf(x)`	Returns `true` if `x` is positive or negative infinity.
`isnan(x)`	Returns `true` if `x` is NaN.
`isnormal(x)`	Returns `true` if `x` is normal, i.e. neither zero, subnormal (denormalized), infinite, nor NaN.
`signbit(x)`	Returns a non-zero 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 NaN.

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.

Fixed-Width Integer Types <cstdint>

The <cstdint> header contains platform-dependent typedefs 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 typedefs 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 hence can 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 next. Functions are only and always used to obtain a T value; whereas Booleans, ints, 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 can 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`, because it cannot represent all values with three 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 are not, and for `char` and `wchar_t` it is unspecified.
`is_integer`	Identifies integer types (includes Booleans and character types).
`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 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. Usually `true` for all standard floating-point types. Using a signaling NaN in operations results in an exception; using a quiet NaN does not.
`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`.

(*continued*)

Member	Description
`denorm_min()`	Returns the 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 that 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, e.g., 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: `pow()`, `sqrt()`, `exp()`, `log()`, and `log10()`, as well as all trigonometric and hyperbolic functions: `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 *modulus* or *magnitude*	$\sqrt{real^2 + imag^2}$
`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)*	$m(\cos\varphi + i\sin\varphi)$
`proj()`	Projection onto the Riemann sphere	$(\infty, \pm 0)$ if infinite *real* or *imag*; else (*real*, *imag*)

Compile-Time Rational Numbers <ratio>

The `std::ratio<Numerator, Denominator=1>` template type 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). Although `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>>`, for instance, to represent a duration of milliseconds, or as `duration<int,ratio<60>>` for a duration of minutes.

Convenience `typedef`s 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 they are 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` `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):

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

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 non-deterministic 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, for example, consists of 625 integers). Still, because they are also the fastest generators, these engines should be your default choice if size is of no concern.
- std::subtract_with_carry_engine: Although an improvement over the linear congruential engines in terms of quality (but not speed), these engines have much lower quality and performance than a Mersenne twister. Their state size is more moderate, though (generally 96 bytes).

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 values 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` numbers generated 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 these:

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 previous 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 standalone (see the 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** Because choosing between all the previous predefined engines can be daunting, an implementation must also offer a `std::default_random_engine` that should be good enough for most applications (it may be a `typedef` for one of the others).

Non-Deterministic Random Number Generator

A `random_device`, in principle, does not generate pseudorandom numbers, but truly *non-deterministic* 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 non-deterministic 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 non-copyable 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 with pseudorandom number engines, though, its `operator()` may throw an `std::exception` if it failed to generate a number (due to hardware failure, for example).

Although 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. Although 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 (for example, at construction time). The recommended way of doing this is with a `random_device`, but as you saw earlier, this may also just generate pseudorandom numbers. A popular alternative is to seed with the current time (see 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. Describing all the math behind them falls outside the scope of this book, but 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 that 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 mantisse.

Bernoulli Distributions

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

Generates random Boolean values with `p` equal to the probably 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.

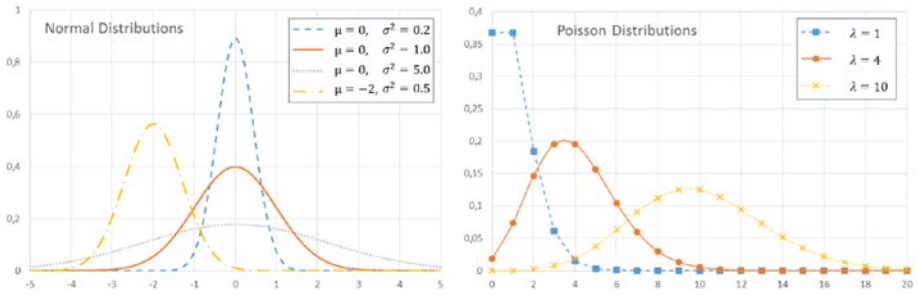


Figure 1-1. *Probability distributions for some example normal and Poisson distributions, plotting the probability (between 0 and 1) that a value is generated*

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 equals 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 that 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 whose weights are 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 whose weights are 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<double> intervals = { 1,20,40,60,80 };
std::vector<double> 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 previous 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.

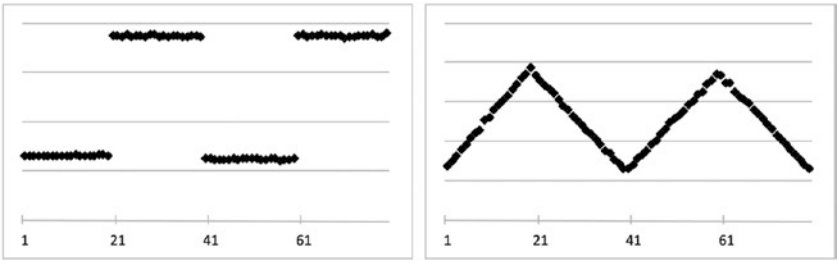


Figure 1-2. *Difference between a piecewise constant and piecewise linear distribution*

The graph on the right shows a `piecewise_linear_distribution` with the same intervals and 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.

If you use a `piecewise_linear_distribution` with intervals of different sizes, the graph is not 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, the value at the beginning of the interval is three times more likely to be generated than the value at the end. Therefore, if the interval is for example twice as long, all probabilities are 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>&)`	Constructor that converts 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 constructors.

Here is an example:

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

(*continued*)

Operation	Description
`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 are zero-initialized for `shift()`, whereas `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()`, ...), 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 indexes. These indexes 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 fewer 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, and they apply the operator to the elements referred to by the `slice_array`. For example:

```
std::valarray<int> 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<int> 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 with `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 set of operations similar to those of a `slice_array`. How the different sizes and strides are used is best explained with an example:

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

This example has two values for size and stride, so the `gslice` creates 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 these 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 has these parameters (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 (that is, 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, though, 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 a set of operations analogous to those of a `slice_array`. Here is an example:

```
std::valarray<int> 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<bool> even = ((ints % 2) == 0);
// Count the number of true values in even. (see Chapter 4)
int count = std::count(begin(even), end(even), true);
// Construct a valarray with count elements of value 4.
std::valarray<int> 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

Finally, 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 a set of operations analogous to those of a `slice_array`. For example:

```
std::valarray<int> ints = { 0,1,2,3,4 };
std::valarray<size_t> indices = { 1,3,4 };
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, it also introduces 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. 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` (see 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. Certain operations, assignments in particular, remain allowed, 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++ Standard Library Quick Reference*,
DOI 10.1007/978-1-4842-1876-1_2

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, effectively 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; --> Undefined behavior: one was moved to other
```

If no move member is defined, either explicitly or implicitly, overload resolution for `T&&` falls back to `T&` or `T`, and in the latter case still creates 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 exceptions 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 from 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** For optimal performance with nontrivial custom types, it is not only crucial to define move members, but also equally important to always do so *with a* `noexcept` *specifier*. The container classes from Chapter 3 extensively use moving to speed up operations such as adding a new element, or when relocating arrays of elements (for example, with sequential containers). Similarly, many algorithms from Chapter 4 benefit if efficient move members are provided (and/or nonmember `swap()` operations, discussed later). However, and especially when moving arrays of elements, these optimizations often take effect only 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 do), 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

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.

Although already quite efficient if proper move members are available, for truly optimal performance you should consider specializing these template functions: for instance, to eliminate the need to move to a temporary. Many algorithms from Chapter 4, for example, 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 specializing 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 for instance `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)` will not use user-defined `swap()` functions outside the `std` namespace, whereas `swap(x,y)` alone will not work *unless* there is such a user-defined function.

Pairs and Tuples

Pairs

<utility>

The `std::pair<T1,T2>` template `struct` is a copyable, moveable, 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 two template type parameters can be deduced automatically using auxiliary function `std::make_pair()`:

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

■ **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`.

Tuples **<tuple>**

`std::tuple` is a generalization of `pair` that allows any number of values to be stored (that is, zero or more, not just two): `std::tuple<Type...>`. It is mostly analogous to `pair`, including the `make_tuple()` auxiliary function. 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:

```
auto t = std::make_tuple(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
```

An alternative way to obtain values of a `tuple` is by unpacking it using the `tie()` function. The special `std::ignore` constant may be used to exclude any value:

```
int one, two; double three;
std::tie(one, two, three, std::ignore) = t;
```

■ **Tip** The `std::tie()` function may be used to compactly implement lexicographical comparisons based on multiple values. For instance, the body of `operator<` for the `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` (see Chapter 3) in their respective headers, but not `tie()`.

A final 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. It is designed to forward arguments (that is, while avoiding copies) to the constructor of a `tuple`, in particular in the context of functions that accept a tuple by value. A function `f(tuple<std::string, int>)`, for instance, can then be called as follows: `f(std::forward_as_tuple("test", 123));`.

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

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==`, and the remaining forward to `operator<`. So, your class only needs to implement `operator==` and `<`, and the others are generated automatically when you add a `using namespace std::rel_ops;`

```
// 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 memory allocated by `new` or `new[]` should be

managed by a smart pointer, or, for the latter, a container such as `vector` (see 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 therefore cannot 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 that a `unique_ptr` can generally 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");
  ...
```

■ **Tip** Using `make_unique()` not only may shorten your code, but also prevents certain types of memory leaks. Consider `f(unique_ptr<X>(new X), g())`. If `g()` throws after the `X` was constructed, but before it was assigned to its `unique_ptr`, the `X` pointer leaks. Writing `f(make_unique<X>(), g())` instead guarantees such leaks do not occur.

Other uses of `unique_ptrs` that make them a truly essential utility include these:

- 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 (such bugs can be subtle sometimes: see the next **Tip**).

- They can be stored safely inside the containers from Chapter 3.
- When used as member variables of another class, they eliminate the need for explicit deletes in their destructor. Moreover, they prevent the compiler from generating error-prone copy members for objects that are supposed to exclusively own dynamic memory.

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 in Chapter 3.

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, whereas the latter replaces it with the given T*. The key difference 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. It 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 previous 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 expressions containing release() subexpressions do not contain any throwing subexpressions as well. In the 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. For legacy APIs, though, 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 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 non-default 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 functor type may be used.

std::auto_ptr

At the time of writing, the `<memory>` header still defines a second smart pointer type for exclusive ownership, namely `std::auto_ptr`. This has been deprecated, however, in favor of `unique_ptr` in C++11, and is set to be removed in C++17. We therefore do not discuss it in detail. Essentially, an `auto_ptr` is a flawed `unique_ptr` that is implicitly moved when copied: this makes them not only error-prone but also dangerous (and in fact illegal) to use with the standard containers and algorithms from Chapter 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()` checks whether the count equals one.

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(). A shared_ptr cannot manage dynamic arrays, though. What really sets it apart is that shared_ptrs 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. To construct a new shared_ptr, it is again recommended to use make_shared(): for the same reasons as with make_unique() (shorter code and memory leak prevention), 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 the 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(), or const_pointer_cast(). If the result is non-null, the reference count is safely 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, and 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<> 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, when 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 non-empty `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. Although 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, because nothing would then prevent the last co-owner from concurrently deleting it. To access the resource, a `weak_ptr` therefore 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
```

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:

```
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 not only can 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, storing or providing callback functions.

This section outlines the functors defined in `<functional>`, as well as its facilities for creating and working with functors.[1] We also briefly introduce lambda expressions, the powerful C++11 language construct for creating functors.

[1] `<functional>` contains many deprecated facilities we do not discuss: `ptr_fun()`, `mem_fun()`, `mem_fun_ref()`, `bind1st()`, and `bind2nd()`, plus their return types, as well as the base classes `unary_function` and `binary_function`. All of these have already been removed from the C++17 version of the standard and should not be used.

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

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&.

Because 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 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 from Chapter 3. In the following example, for instance, you could not declare a vector<int&>, because int& values cannot be assigned to:

```
int i = 234;
std::vector<std::reference_wrapper<int>> v{ std::ref(i) }; // cf. Chapter 3
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 this section's introduction:

- 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(begin(array), end(array), std::greater<int>());
```

As of C++14, all of these 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*, because their function call operator conveniently deduces

the parameter type. In the previous `sort()` example, for instance, you could simply use `std::greater<>`. The same functor can even be used for different types:

```
std::plus<> functor;                              // defaults to std::plus<void>
std::cout << functor(234,432) << ' ' << functor(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.

Passing a unary/binary functor, `predicate`, to `std::not1()/not2()` creates a new functor (of type `unary_negate/binary_negate`) that negates `predicate`'s result (that is, evaluates to `!predicate()`). For this to work, the type of `predicate` *must* define a public member type `argument_type`. All functor types in `<functional>` have this.

Generic Function Wrappers

The `std::function` template class is designed for wrapping a copy of any kind of *callable* entity: that is, any kind of function object or pointer. This includes the results of, for example, `bind` or lambda expressions (both explained in more detail shortly):

```
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 as 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 from Chapter 4, it is generally unspecified how often they copy their arguments. So to guarantee no copies are made, you must use `(c)ref()`.

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`, and so forth) 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 placeholders is implementation specific; and the type of the returned functors is unspecified. Some examples will 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 while 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 see chapters 3, 4, and 6)
   std::vector<char> 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;
}
```

Functors for Class Members

Both `std::function` and `bind()`, introduced earlier, 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 `bind()` and `mem_fn()`, but not `std::functions`, may also be called with a pointer or one of the standard smart pointers (see the previous section) as the first argument (that is, without dereferencing). Interesting also about the `bind()` option is that it can bind the target object itself (see `b_call_fun_on_s`). If that is not required, `std::mem_fn()` generally results in the shortest code because 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
```

LAMBDA EXPRESSIONS

Although not part of the Standard Library, lambda expressions are such a powerful tool for creating functors that they are well worth a short introduction. In particular, when combined with the algorithms from Chapter 4, the concurrency constructs from Chapter 7, and so on, they often form the basis of very expressive, elegant code. Several examples of lambda expressions can be found throughout this book, especially in Chapter 4.

A lambda expression is often said to create an *anonymous function*, but actually it creates a *functor* of unspecified type, also called a *closure*. A lambda expression does not have to start from an existing function like the `<functional>` constructs: the body of its closure's function call operator may contain arbitrary code.

The basic syntax of a lambda expression is as follows:

```
[CaptureBlock](Parameters) mutable -> ReturnType {Body}
```

`Capture block`: Specifies which variables from the enclosing scope to capture. Essentially, the created functor has, for each captured variable, a member with the same name containing a copy of this captured variable if it is *captured by value*, or a reference to it if *captured by reference*. As such, these variables become available for use in the body. The basic syntax of a capture block:

- `[]` captures no variables (cannot be omitted).
- `[x, &y]` captures `x` by value and `y` by reference.
- `[=, &x]` captures all variables from the enclosing scope by value except `x`, which is captured by reference.
- `[&, x,y]` captures all variables by reference except `x` and `y`, which are captured by value.
- `[this]` captures the `this` pointer, granting the body access to all members of the surrounding object.

`Parameters`: Parameters to be passed when calling the functor. Omitting is equivalent to specifying an empty list `()`. The type of parameters may be `auto`.

`mutable`: By default, the function call operator of a lambda functor is always marked as `const`, implying that variables captured by value—that is, copied in member variables—cannot be modified (assigned to, non-`const` members called, and so on). Specifying `mutable` makes the function call operator non-`const`.

`Return type`: The type of the returned value. May be omitted as long as all return statements of the body return the exact same type.

`Body`: The code to execute when the lambda functor is called (non-optional).

It is also possible to specify `noexcept` and/or attributes (after the optional `mutable`), but these are rarely used.

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 non-empty 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 from Chapter 3, for instance, have initializer-list constructors to initialize them with a list of values.

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, you have to use the C-style date and time types and functions defined in `<ctime>`. The `system_clock` from `<chrono>` allows for interoperability with `<ctime>`.

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 one second:

```
using namespace std::chrono;
typedef duration<int, std::ratio<3600>> hours_t;
typedef duration<int64_t, std::milli> millisecs_t; // milli==ratio<1,1000>
const hours_t one_hour(1);
const millisecs_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 `typedefs` 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 1,000 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 has 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 that you 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))
```

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 is not allowed, nor is subtracting one from a `Duration`:

```
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 is allowed as long as no truncation is required; otherwise, `time_point_cast()` can be used:

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

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 that 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_point`s 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; and `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, zero 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. Although 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, and so on).
`time()`	Current point in time as a `time_t`, or `null` on failure. A `time_t*` argument must be passed: if not `null`, value is written there as well.
`difftime()`	The difference between two `time_t`s as a `double` value denoting a time in seconds (result may be negative).

(*continued*)

Function	Returns
`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 `null` 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*` or, respectively, `time_t*` is written, using a *fixed, locale-independent* format. They are thus both limited and *not thread-safe*, so they have been *deprecated* in favor of, for example, `strftime()`.
`strftime()`	Explained next.

Consult your implementation's documentation for safer alternatives for `localtime()` and `gmtime()` (such as `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 equals 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 that 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]`

(*continued*)

Specifier	Output	Range or Example
%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.	2015
%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 / 15 / 15
%D / %F	Equivalent to "%m/%d/%y" / "%Y-%m-%d".	10/21/15 / 2015-10-21
%c / %x / %X	Preferred date + time / date / time representation.	*(see below)*
%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 (see Chapter 6). When executed with a French locale, for example, the output for the previous 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` (`%EC`, `%Ec`, and so on); 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 generally need to output calendar dates and time in C++-style is therefore `localtime()` (to convert a `system_clock`'s `time_t` to `tm`).

C-Style File Utilities

<cstdio>

The next version of the C++ Standard Library is expected to include a more powerful C++-style file system library. For now, this limited set of C-style functions in the <cstdio> header are the only portable file utilities available in the Standard:

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 with the C-style I/O functions briefly discussed in Chapter 5. The temporary file is automatically deleted when it is closed. Returns `nullptr` when the file could not be created.
`char* tmpnam(char*)`	Creates a unique, non-existing filename. If a `char*` argument is provided, the result is stored in this pointer and the 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.

Type Utilities

Runtime Type Identification

<typeinfo>, <typeindex>

The C++ `typeid()` operator is used to obtain information about 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:

```
std::string s;
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>>`" (see Chapter 6), but for other implementations it might just as well be "`Ss`".

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

Because `type_info`s cannot be copied, they cannot be used as keys for the associative arrays from 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* is a construct used to obtain compile-time information on a given type or to transform a given type or types to a related one. Type traits are generally used to inspect and manipulate template type arguments when writing generic code.

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

Type Classification

Each type in C++ belongs to exactly one of *14 primary type categories*. In addition to 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`. Some examples follow (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_class<std::is_class<bool>>::value << '\n';     // true
std::cout << std::is_function<int(void)>::value << '\n';            // true
std::cout << std::is_function<decltype(main)>::value << '\n';       // true
std::cout << std::is_pointer<decltype(&main)>::value << '\n';       // true
struct A { void f() {}; };
void(A::* p)() = &A::f;
std::cout << std::is_member_function_pointer<decltype(p)>() <<'\n';// true
```

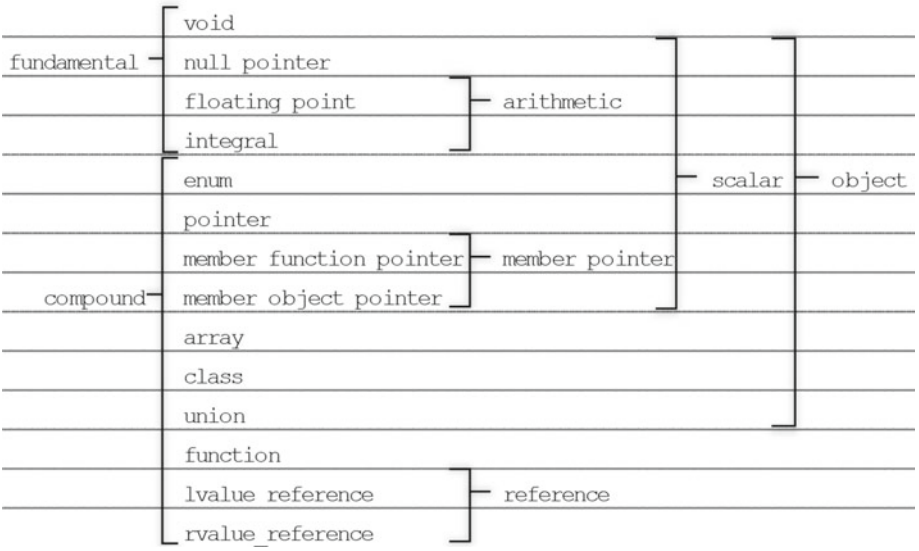


Figure 2-1. *Overview of the type classification traits. The second column lists the 14 primary categories; the other names are those of the composite categories.*

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 one, `has_virtual_destructor`, again have names of the form `is_`*property*.

The following values for *property* 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 non-class types as well (because 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&>`, and so on).
- All the previous *property* names may further be prefixed with either `trivially` or `nothrow`. For instance: `trivially_destructible`, `nothrow_constructible`, or `nothrow_move_assignable`.

The `nothrow` properties hold if the construction, assignment, or destruction 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 (that is, 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 essentially hold 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 non-default 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 (that is, no polymorphism, limited multiple-inheritance, and so on). This is for compatibility with C, because such casts (with C `structs` then) are common practice in C code.
- **`pod`** (plain old data), if `trivial` and `standard_layout`.
- **`literal_type`**, if values may be used in `constexpr` expressions (that is, can be evaluated statically without side effects).
- **`empty`**, for non-polymorphic classes without non-static member variables.

Type Property Queries

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

- **std::alignment_of<T>**: value of the alignof(T) operator
- **std::rank<T>**: array dimensions, such as rank<int>() == 0, rank<int[]>() == 1, rank<int[][5][6]>() == 3, and so on
- **std::extent<T,N=0>**: number of elements of the Nth array dimension, or 0 if unknown or invalid; for instance, extent<int[]>() == 0 and extent<int[][5][6], 1>() == 5.

Type Comparisons

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

Type Transformations

Most type transformation traits are again fairly similar, except that they have no value, but instead a nested typedef named 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 is shown shortly as an example.
- **std::make_y** with **y** either signed or unsigned. If applied on an integral type T, type is a signed or, respectively, unsigned integer type with sizeof(type) == sizeof(T).
- **std::result_of**, defined only for functional types, gives the return type of the function.
- **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.

The header also contains two utility traits to help with type metaprogramming. Their basic use is illustrated by means of a few examples.

- **`std::conditional<B,T1,T2>`** has `type T1` if the `constexpr B` evaluates to `true` and `type T2` otherwise.
- **`std::enable_if<B,T=void>`** has `type T`, but only if the `constexpr B` evaluates to `true`. Otherwise, `type` is not defined.

For all traits of this subsection, a convenience type with name `std::`*trait*`_t<T>` exists defined as `std::`*trait*`<T>::type`. The upcoming example, for instance, shows how convenient `enable_if_t<>` is compared to the full expression.

This first example shows how to use the C++ *SFINAE* idiom to conditionally add or remove functions from overload resolution. SFINAE is an acronym for *Substitution Failure Is Not An Error* and exploits the fact that failure to specialize a template does not constitute a compile error. In this case, it is the absence of the `type typedef` that causes substitution to fail:

```
// 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<T>::value>
copy(T(&from)[N], T(&to)[N])
{ for (size_t i = 0; i < N; ++i) to[i] = from[i]; }
```

A second 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:
   typedef remove_reference_t<T> U;
public:
   typedef conditional_t<is_array<U>::value, remove_extent_t<U>*,
           conditional_t<is_function<U>::value, add_pointer_t<U>,
           remove_cv_t<U>>> type;
};
```

CHAPTER 3

Containers

The C++ Standard Library provides a selection of different data structures called *containers* that you can use to store data. 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 containers. Because this book is a quick reference, it is impossible to discuss all containers in depth. The `std::vector` container is explained in more detail compared to the others. Once you know how to work with one container, you know how to work with others.

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 (R):* A bidirectional iterator that support jumping to elements at arbitrary indexes

P. Van Weert and M. Gregoire, *C++ Standard Library Quick Reference*,
DOI 10.1007/978-1-4842-1876-1_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 to, 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`	Calculate 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 iterator and can therefore be used with the algorithms from Chapter 4 as well. Also, certain containers, sequential containers in particular, likely define their iterators as `typedef`s for regular pointers. 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

An iterator tag is an empty type used to differentiate between the different iterator categories seen earlier. The Standard defines `std::`*category*`_iterator_tag` types for the following values of *category*: `forward`, `bidirectional`, and `random_access`. The type trait expression `std::iterator_traits<Iter>::iterator_category` evaluates to the iterator tag type of the given iterator type `Iter`. This can be used by generic algorithms to optimize their implementation based on the category of its iterator arguments. For example, the `std::distance()` method explained in an upcoming section uses the iterator tag to choose between an implementation that linearly calculates the distance between two iterators and a more efficient one that simply subtracts two iterators.

If you implement your own iterators, you should therefore specify its tag. You can do this either by adding a `typedef` *Tag* `iterator_category` to your implementation, where *Tag* is one of the iterator tags, or by specializing `std::iterator_traits` for your type to provide the correct tag type.

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 or `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 };
auto beginIter = std::cbegin(myArray);
auto endIter = std::cend(myArray);
for (auto iter = beginIter; iter != endIter; ++iter) {
   std::cout << *iter << std::endl;
}
```

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 also earlier), the implementation automatically falls back to a method that works for that iterator (for example, 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)`: Equivalent to `advance(iter, dist)` and returns `iter`.
- `std::prev(iter, dist)`: 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`. At the end is a reference with all available methods supported by these 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, 12);
```

This creates `myVector` containing `100` elements with value `12`. The second parameter is optional. If you omit it, new elements are zero-initialized, which is `0` for the case of integers.

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.

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, which throws an `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.

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<int> 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<int> 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). If you do use a loop, make sure you do not cache the end iterator, because `insert()` might invalidate it.

`insert()` can also be used to insert a range of elements anywhere in the `vector` or to concatenate (append) two vectors. When using `insert()`, you do not have to resize the `vector` yourself. For example:

```
std::vector<int> v1{ 1,2,3 };
std::vector<int> 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 `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()`. Suppose you have a `vector` of `Person` objects. You can add a new person at the back in these two similar ways:

```
persons.push_back(Person("Sheldon", "Cooper"));
persons.emplace_back("Leonard", "Hofstadter");
```

The arguments to emplacement functions are perfectly 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 or may even be the only way to add elements if they 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, returned by `size()`, which is the number of elements contained in the `vector`. Use `empty()` to check whether a `vector` is empty or not. Take care, though, not to confuse `empty()` with `clear()`: the former returns a Boolean, and the latter removes all elements.

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 are going to add, it is crucial for performance to preallocate sufficient capacity to avoid reallocation. Failure to do so will cause a significant performance hit. This can be done 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 non-empty `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, though, which may be ignored by an implementation: for example, for performance reasons.

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

The formerly empty container is then destroyed, containing all elements, leaving the original one empty. This idiom is often also written more briefly as follows:

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

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. The following example removes all elements equal to 2 from a vector:

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

If you do use a loop as shown here, make sure you do not cache the end iterator because `erase()` will invalidate it. To avoid this and other mistakes, it is always recommended that you use standard algorithms instead of hand-written loops. When you want to remove multiple elements, you can use the remove-erase idiom. This pattern first uses the `std::remove()` or `std::remove_if()` algorithm. As Chapter 4 explains, 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<int> 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 (that is, those to remove) can be different depending on your compiler.

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** In the remove-erase idiom, do not forget to specify the end iterator as second parameter to `erase()`, as marked in bold in the previous examples. Otherwise you will only delete one element!

---

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, for example, 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>

A `deque` is a double-ended queue, a container similar to a `vector` that supports efficient insertion and deletion both at the beginning and at the end. The Standard does not require `deque` elements to be stored contiguously in memory, so reallocation done by a `deque` may be cheaper than for a `vector`. Nevertheless, `deque` supports random access and random-access iterators.

The operations on a `deque` are almost the same as for a `vector`, with a few minor differences. A `deque` does not have the concept of capacity because it does not have to store its elements contiguously, so none of the methods related to capacity are available. Moreover, a `deque` provides a `push_front()` and `pop_front()` in addition to `push_back()` and `pop_back()`.

Here is an example of using a `deque`:

```
std::deque<int> myDeque = { 1,2,3,4 };
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*: Amortized constant O(1) at the beginning and end; otherwise linear in the distance from the insertion point to the beginning or end O(N)
- *Deletion*: O(1) at the beginning or end; otherwise linear in the distance to the beginning or end O(N)
- *Access*: O(1)

std::array <array>

An `array` is a container with a fixed size specified at compile time as a template argument, supporting random-access iterators. For an `array`, both `size()` and `max_size()` return the same.

The following defines an array of three integers:

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

These integers are *uninitialized*. This is different from all other containers, which 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
```

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

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

For arrays, this may be more efficient than the generic `std::fill()` algorithm explained in Chapter 4.

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. Both therefore store elements non-contiguously in memory.

A first downside is that random access therefore is not possible in constant time. Because of this, `operator[]` is not supported. To access a specific element, 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, though, insertion and deletion at that place are efficient because they only need to modify a couple of links.

A second downside is that elements may become scattered in memory, which is bad for locality and hurts performance due to an increased number of cache misses.

■ **Tip** Because of the aforementioned downsides, only use a `list` or `forward_list` instead of a `vector` if a profiler shows that it is more efficient for your use case.

The operations supported by `list` and `forward_list` are 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.

Complexity

Both `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<int> list1{ 1,7,5 }, list2{ 5,6,2 }, list3{ 3,4 };
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, an `std::out_of_range` exception is thrown.
`data()`	■	□	■	□	□	Returns a pointer to the data of the `vector` or `array`. This is useful to pass the data to legacy C-style array APIs. In older code, you often see the equivalent `&myContainer[0]`.
`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

All sequential containers support the following non-member functions:

Operation	Description
`==`, `!=`, `<`, `<=`, `>`, `>=`	Compares values in two containers (lexicographically)
`std::swap()`	Swaps the contents of two containers

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");
```

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, respectively, not equal.
`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, respectively, stack:

```
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[1]	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.

[1]The way `operator<` is defined for `Person` in the Introduction chapter 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.

Ordered Associative Containers

std::map and std::multimap

<map>

A map is a data structure that stores key-value pairs, using the pair utility class explained in Chapter 2. The elements are ordered according to the key. That is, when iterating over all elements contained in an ordered associative container, they are enumerated in an order with increasing key values, not in the order these elements were inserted. For a map there can be no duplicate keys, whereas a multimap supports duplicate keys.

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:

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

Iterators for a map<Key,Value> or multimap<Key,Value> are bidirectional and point to a pair<Key,Value>. For example:

```
auto iter = begin(myMap);   // the type of *iter is pair<Person, int>
std::cout << "Key=" << iter->first.GetFirstName(); // Key=Bart (not Jenne)
std::cout << ", Value=" << iter->second;           // , Value=2
```

operator[] can be used to access elements in a map. If a requested element does not exist, it is default constructed, so it can also be used to insert elements:

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

You can add more elements to the map with insert():

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

There are several versions of the insert() method:

```
std::pair<iterator, bool> insert(pair)
```

Inserts the given key-value pair. A pair is returned with an iterator pointing to either the inserted element (a key-value pair) or the already-existing element, and a Boolean that is true if a new element was inserted or false otherwise.

```
iterator insert(iterHint, pair)
```

Inserts the given key-value pair. An implementation may use the given hint to start searching for an insertion position. An iterator is returned that points to the inserted element or to the element that prevented the insertion.

`void insert(iterFirst, iterLast)`

Inserts the key-value pairs from the range [`iterFirst`, `iterLast`).

`void insert(initializerList)`

Inserts the key-value pairs from the given `initializer_list`.

There is also an `emplace()` method that allows you to construct a new key-value pair in place. It returns a `pair<iterator, bool>` similar to the first `insert()` method in the previous list. For example:

```
myMap.emplace(Person("Anna"), 4);
```

To avoid the creation of all temporary objects, though, you must use so-called *piecewise construction*, as explained in the section on `pairs` in Chapter 2:

```
myMap.emplace(std::piecewise_construct,
              std::forward_as_tuple("Anna"), std::forward_as_tuple(4));
```

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. The `insert()` method takes a single key instead of a `pair`. For example:

```
std::set<int> mySet{ 3,2,1 };
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`.

An iterator for a `set` or `multiset` is bidirectional and points to the actual key, not to a `pair`, as is the case for `map` and `multimap`. Keys are always sorted.

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.

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. Unless a concrete `Compare` functor instance is passed to the container's constructors, one is default-constructed. Here are the more complete template definitions of all ordered associative containers:

```
template<class Key, class Value, class Compare = std::less<Key>>
class map;
template<class Key, class Value, class Compare = std::less<Key>>
class multimap;

template<class Key, class Compare = std::less<Key>>
class set;
template<class Key, class Compare = std::less<Key>>
class multiset;
```

■ **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 heterogeneous lookups. A classic example is lookups with string literals for `std::string` keys: `std::less<>` then avoids the creation of temporary `std::string` objects. A `set` with `string` keys and a transparent operator functor, for instance, is declared as follows: `std::set<std::string, std::less<>> mySet;`.

Complexity

The complexity for all four ordered associative containers is the same:

- *Insertion*: O(log(N))
- *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, an `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()`. For `map` or `set`, this range can only contain 0 or 1 elements.

Modifiers

All associative containers support the following methods:

Operation	Description
`clear()`	Clears the container.
`emplace()`	Constructs a new element in place.
`emplace_hint()`	Constructs a new element in place. An implementation may use the given hint to start searching for the insertion position.
`erase()`	Removes an element at a specific position, a range of elements, or all elements matching a given key.
`insert()`	Inserts new elements.
`swap()`	Swaps the contents of two containers.

Observers

All ordered associative containers support the following observers:

Operation	Description
`key_comp()`	Returns the key compare functor
`value_comp()`	Returns the functor used to compare key-value pairs based on their keys

Non-Member Functions

All ordered associative containers support the same set of non-member functions as the sequential containers: `operator==`, `!=`, `<`, `<=`, `>`, `>=`, and `std::swap()`.

Unordered Associative Containers

`<unordered_map>`, `<unordered_set>`

There are four unordered associative containers: `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset`. They are similar to the ordered associative containers (`map`, `multimap`, `set`, and `multiset`), except that they do not order the elements but instead store them in buckets in a hash map. The interfaces are similar to the corresponding ordered associative containers, except that they expose hash-specific interfaces related to the hash policy and buckets.

Hash Map

A *hash map* or *hash table* is an efficient data structure storing its elements in buckets.[2] 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 template definitions for all unordered associative containers:

```
template<class Key, class Value, class Hash = std::hash<Key>,
   class KeyEqual = std::equal_to<Key>> class unordered_map;
template<class Key, class Value, class Hash = std::hash<Key>,
   class KeyEqual = std::equal_to<Key>> class unordered_multimap;

template<class Key, class Hash = std::hash<Key>,
   class KeyEqual = std::equal_to<Key>> class unordered_set;
template<class Key, class Hash = std::hash<Key>,
   class KeyEqual = std::equal_to<Key>> class unordered_multiset;
```

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. Details of writing proper hash functions fall outside the scope of this book.

The Standard provides the following `std::hash` template (the base template is defined in `<functional>` but is included also in the `<unordered_xxx>` headers):

```
template<class 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.

[2]Technically, you could easily implement a hash map without buckets: for example, 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 following is an example of how you could implement a `std::hash` specialization for the `Person` class defined in the introduction chapter. 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. Simply XORing values generally does not give sufficiently randomly distributed integers, but if both operands are already hashes, it can be considered acceptable:

```
namespace std {
   template<> struct hash<Person> {
      // Two nested types required by the Standard for specializations, but
      // not specified to be part of the (unspecified) hash<Key> template:
      typedef Person argument_type;
      typedef std::size_t result_type;

      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;
      }
   };
}
```

■ **Note** Although adding types or functions to the `std` namespace is generally disallowed, adding 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

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 `unordered_map` (UM), `unordered_multimap` (UMM), `unordered_set` (US), and `unordered_multiset` (UMS), 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)`	Returns an iterator to the first or one past the last element in the bucket with given index
`bucket()`	Returns the index of the bucket for a given key
`bucket_count()`	Returns the number of buckets
`bucket_size()`	Returns the number of elements in the bucket with a given index
`cbegin(int)` `cend(int)`	`const` versions of `begin(int)` and `end(int)`
`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

All unordered associative containers only support `operator==`, `operator!=`, and `std::swap()` as non-member functions.

Allocators

All containers except `array` and `bitset` support another template type parameter we have not shown yet—one that allows you to specify an allocator type. This always has a default value, though, and you should normally simply ignore it. It is there in cases when you want to have more control over how memory for the container is allocated. So, in theory, you could write your own allocator and pass it to the container. This is an advanced topic that falls outside the scope of this book.

For example, the complete definition of the `vector` template is as follows:

```
template<class T, class Allocator = allocator<T>>
class vector;
```

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++ Standard Library Quick Reference*,
DOI 10.1007/978-1-4842-1876-1_4

Algorithms <algorithm>

This section gives an overview of all available algorithms, organized into subsections according to functionality. All algorithms are defined in the `<algorithm>` header file unless otherwise noted.

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 often accept a *callable* as one of their parameters: for example, a unary or binary operation or predicate. This callable can be a lambda expression, a function object, or a function pointer. Lambda expressions and function objects are discussed in Chapter 2.

General Guidelines

First, whenever possible, use standard algorithms instead of self-written loops, 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.

Second, for several algorithms, certain containers offer equivalent specialized member functions (see Chapter 3). These are more efficient and should therefore be preferred over the generic algorithms. In the algorithm descriptions that follow, we always list these alternatives.

Finally, many of the algorithms move or swap elements. If no implicit or explicit move and/or swap functions are available, these algorithms fall back to copying elements. For optimal performance, you should therefore always consider implementing specialized move and/or swap functions for nontrivial custom data types. Types offered by the Standard Library always provide these where appropriate. We refer to Chapter 2 for more information regarding move semantics and swap functions.

Applying a Function on a Range

```
Function for_each(InIt first, InIt last, Function function)
```

> Calls the given function for each element in the range `[first, last)`, and returns `std::move(function)`. Note that when iterating over an entire container or C-style array, a range-based `for` loop is more convenient.

```
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`, which is allowed to be equal to `first1` or `first2` to perform an in-place transformation. For the first version, a unary operation is performed on each transformed element. For the second, a binary operation is performed on each transformed element with the corresponding element from the second range. Let *length* = `(last1 - first1)`, then the binary operation is executed on pairs `(*(first1 + n), *(first2 + n))` with $0 \leq n < length$. Returns the end iterator of the target range, so `(target + length)`.

Example

The following example uses `transform()` to double all the elements in a `vector` using a lambda expression, then uses `transform()` to negate the elements using a standard function object, and finally outputs all the elements to the console using `for_each()`. This code snippet additionally needs `<functional>`:

```
std::vector<int> 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(cbegin(vec), cend(vec),
   [](auto& element) { std::cout << element << " "; });
```

The output is as follows:

```
-2 -4 -6 -8 -10 -12
```

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 ordered and unordered 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<int> vec{ 1,2,3,4,5,6 };
bool allOf = std::all_of(cbegin(vec), cend(vec),
   [](auto& element) { return element % 2 == 0; });  // false
```

Finding Elements

```
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 ordered and unordered 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"; });
```

Binary Search

All of the following algorithms require that the given range [first, last) is sorted or at least partitioned on value (partitioning is explained later). 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 ordered and unordered associative containers 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<int> 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
```

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<int> 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

All the 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])
```

For `search()`/`find_end()`, respectively, returns an iterator to the beginning of the first/last 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 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.

Min/Max

```
constexpr const T& min(const T& a, const T& b[, Compare comp])
constexpr const T& max(const T& a, const T& b[, 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> t[, Compare comp])
constexpr T max(initializer_list<T> t[, Compare comp])
```

Returns a copy of the minimum or maximum value in a given `initializer_list`, or a copy of the leftmost element if there are several elements equal to this extreme.

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

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

```
constexpr pair<T, T> minmax(initializer_list<T> t[, 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 elements are equal to the maximum, then a copy of the rightmost is returned.

```
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, respectively, 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.

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) pair-wise match. The second range must have at least n elements. The four-argument version discussed later 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 pair-wise match. The second range must have at least n elements. The following four-argument version 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)`.

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).

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
```

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 passed 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(), 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 predicate)
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 the end iterator of 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<int> 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(), 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 the end iterator of 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<int> 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, respectively, outFalse. 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(), 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)).

```
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, though. It is, however, (non-stably) partitioned on the element nth points to.

```
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)`.

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<int> 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
```

Shuffling

```
void shuffle(RanIt first, RanIt last, UniformRanGen generator)
```

Shuffles the elements in the range `[first, last)` using randomness generated by a uniform random-number generator. The random-number-generation library is explained in Chapter 1.

```
void random_shuffle(RanIt first, RanIt last[, RNG&& rng])
```

Deprecated in favor of `shuffle()`, but mentioned for completeness. It shuffles the elements in the range `[first, last)`. The random number generator, `rng`, is a functor whose function call operator accepts an integral parameter `n` and returns an integral random number in the range `[0, n)` with `n > 0`. If no random-number generator is provided, the implementation is free to decide how it generates random numbers.

Example

The following example shuffles the elements in a `vector`. 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> vec{ 1,2,3,4,5,6 };
std::shuffle(begin(vec), end(vec), gen);   // Possible result: 5 1 4 2 6 3
```

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.

```
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. *[Alternatives: `list::merge()`, `forward_list::merge()`.]*

```
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.

```
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 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 in the target range are sorted. The target range must be big enough to accommodate the elements of the set operation. The input and output ranges are not allowed to overlap. Returns the end iterator of the constructed target range.

- *Union*: All elements of both input ranges. If an element is in both input ranges, it appears only once in the output range.
- *Intersection*: All elements that are in both input ranges.
- *Difference*: All elements that are in `[first1, last1)` and that are not in `[first2, last2)`.
- *Symmetric difference*: All elements that are in `[first1, last1)` and that are not in `[first2, last2)`, and all elements that are in `[first2, last2)` and that are not in `[first1, last1)`.

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 adapter discussed in the previous chapter is 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 way. Although the exact ordering is implementation-specific, it must satisfy the following key properties: no element is greater than its first element, and both removing this greatest element and adding any new element can be done 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>

The following algorithms are defined in the <numeric> header:

```
T accumulate(InIt first, InIt last, T startValue[, BinOp op])
```

Returns result, which is calculated by starting with result equal to startValue and then executing result += element or result = op(result, element) for each element in the range [first, last).

```
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 += (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.

```
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.

```
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`.

Example

The following code snippet uses the `accumulate()` algorithm to calculate the sum of all elements in a sequence:

```
std::vector<int> vec{ 4,2,5,1,3,6 };
int sum = std::accumulate(begin(vec), end(vec), 0);     // 21
```

The `inner_product()` algorithm can be used to calculate the so-called *dot product* of two mathematical vectors:

```
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
```

Iterator Adaptors **<iterator>**

The Standard Library provides the following iterator adaptors:

- **`reverse_iterator`**: Reverses the order of the iterator being adapted. Use `make_reverse_iterator(Iterator iter)` to construct one.
- **`move_iterator`**: Dereferences the iterator being adapted as an rvalue. Use `make_move_iterator(Iterator iter)` to construct one.
- **`back_insert_iterator`**: An iterator adaptor that inserts new elements at the back of a container using `push_back()`. Use `back_inserter(Container& cont)` to construct one.

- **front_insert_iterator**: An iterator adaptor that inserts new elements at the front of a container using `push_front()`. Use `front_inserter(Container& cont)` to construct one.
- **insert_iterator**: An iterator adaptor that inserts new elements in a container using `insert()`. To construct one, use `inserter(Container& cont, Iterator iter)`, where `iter` is the insertion position.

The following example copies all elements from a `vector` to a `deque` in reverse order by using a `front_insert_iterator` adaptor on the `deque`. Next, it concatenates all `strings` in the `vector` using `accumulate()` (whose default combining operator, `+`, performs concatenation for `strings`). Because `move_iterator` adaptors are used here, the `strings` are moved rather than copied from the `vector`:

```
std::vector<std::string> v{ "1","2","3" };

std::deque<std::string> reversed;
std::copy(cbegin(v), cend(v), std::front_inserter(reversed));

std::string s123 = std::accumulate(std::make_move_iterator(begin(v)),
                                   std::make_move_iterator(end(v)),
                                   std::string());
std::cout << s123 << '\n';                              // 123
```

CHAPTER 5

Stream I/O

The C++ stream-based I/O library allows you to perform I/O operations without having to know details about the target to or source from which you are streaming. A stream's target or source could be a string, a file, a memory buffer, and so on.

Input and 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.

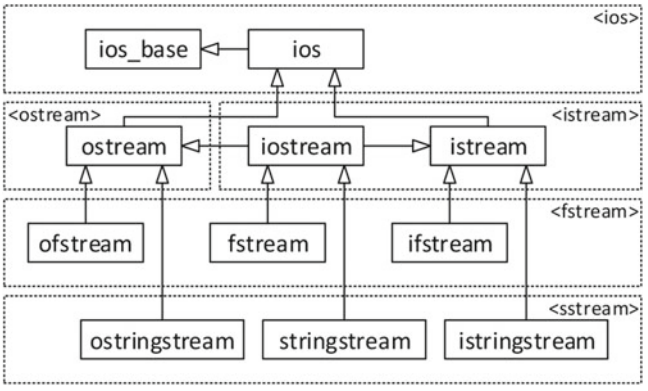


Figure 5-1. *The hierarchy of stream-related classes*

More accurately, the library defines templates called `basic_ios`, `basic_ostream`, `basic_istringstream`, and so on, all templated on a character type. All classes in the hierarchy, except `ios_base`, are `typedefs` for these templated classes with `char` as template type. For example, `std::ostream` is a `typedef` for `std::basic_ostream<char>`. There are equivalent `typedefs` for the `wchar_t` character type called `wios`, `wostream`, `wofstream`, and so on. The remainder of this chapter only uses the `char` `typedefs` shown in Figure 5-1.

P. Van Weert and M. Gregoire, *C++ Standard Library Quick Reference*,
DOI 10.1007/978-1-4842-1876-1_5

In addition to the headers in the figure, there is also <iostream>. Somewhat confusingly, this does not really define std::iostream itself, because this is done by <istream>. Instead, <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. The latter two are intended for output of errors and logging information, respectively. Their destinations are implementation specific.

The library also provides the std::basic_streambuf, basic_filebuf, and basic_stringbuf templates and their various typedefs, 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. They are briefly discussed toward the end of this chapter.

The header <iosfwd> contains forward declarations of all the standard I/O library types. It is useful to include it in other header files without having to include the complete template definitions of all the types you require.

Helper Types <ios>

The following helper types are defined in <ios>:

Type	Description
std::streamsize	A typedef 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	A typedef 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 typedefs are provided: std::streampos and wstreampos for the character types char and wchar_t.

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:

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 Table 5-1 and Table 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.

It is also possible to modify flags by streaming I/O manipulators, discussed in the next section.

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. The third flag, `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 the output to be flushed after each output operation.
`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`

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.
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()	Reads or writes quoted strings and properly handles embedded quotes. An example of this manipulator is given in the section on how to implement your own operator<< and operator>> later in this chapter.

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:

```
Left:     $1.23__
Right:    __$1.23
Internal: 0x___7b
```

std::ios <ios>

The ios class defined in <ios> inherits from ios_base and provides 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.

The following state-related methods are provided:

Method	Description

`good()`
`eof()`
`bad()`
`fail()`
	Returns `true` if, respectively, the `badbit`, `failbit`, and `eofbit` are
	not 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()`.
`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())`.

Besides these state-related methods, the following additional ones are defined by `ios`:

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 everything from another `ios` instance except its state.
`tie()`	Ties any output stream to the `this` stream, which means the tied output stream is flushed each time an input or output operation is performed on the `this` stream.
`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.

The default initialization of `std::ios` has the following effect:

- Flags are set to `skipws | dec`.
- Precision is set to 6.
- The field width is set to 0.
- The fill character is set to `widen(' ')`.
- The state is set 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 exceptions 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 (explained in detail later in this chapter). No exceptions are thrown; only the fail bit of the stream is set to 1:

```
std::ifstream ofs("non_existing_file.ext");
std::cout << ofs.fail() << std::endl;        // 1
```

If you want to use exceptions instead, the code can be rewritten as follows:

```
std::ifstream ofs("non_existing_file.ext");
try {
   ofs.exceptions(ofs.failbit);      // Enable exceptions for failures.
} catch (const std::ios_base::failure& exception) {
   std::cout << exception.what() << std::endl;
}
```

A possible output could be

```
ios_base::failbit set: iostream stream error
```

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` 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.

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

`<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.

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)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 `ostreams` 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.

`<istream>` also defines the following extra I/O manipulator:

Manipulator	Description
`ws`	Discards any whitespace currently in the stream.

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.

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.

Example

```
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 strings. 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 object
- `void str(string&)`: Sets the underlying string object to a copy of the given one

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&`.
- `[i|o]fstream([i|o]fstream&&)`: Move constructor.

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

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) {
   fs.clear();                              // First clear any error states.
   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);
}
```

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;
```

operator<< and >> for Custom Types

You can write your own versions of the stream output and extraction operators `operator<<` and `operator>>`. Here is an example of both operators for the `Person` class, using the `std::quoted()` manipulator to handle spaces in names:

```
std::ostream& operator<<(std::ostream& os, const Person& person) {
   os << std::quoted(person.GetFirstName()) << ' '
      << std::quoted(person.GetLastName());
   return os;
}

std::istream& operator>>(std::istream& is, Person& person) {
   std::string firstName, lastName;
   is >> std::quoted(firstName) >> std::quoted(lastName);
   person.SetFirstName(firstName); person.SetLastName(lastName);
   return is;
}
```

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"
```

Stream Iterators <iterator>

The `<iterator>` header defines two stream iterators, `std::istream_iterator` and `std::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<double> 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.

This second example uses both an `istream_iterator` to read an unspecified number of `doubles` from the console and an `ostream_iterator` to write the read `doubles` 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>

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 typedefs are provided, `std::streambuf` and `std::wstreambuf`, where the template type is, respectively, `char` or `wchar_t`. File streams use `std::(w)filebuf`, and string streams use `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.

C-Style Output and Input <cstdio>

In addition to the file utilities explained in Chapter 2, the `<cstdio>` header also defines the C-style I/O library, including functions for character-based I/O (`getc()`, `putc()`, ...) and formatted I/O (`printf()`, `scanf()`, ...). All the C-style I/O functionality is subsumed by the type-safe C++ streams, which also have better-defined, portable error

handling.[1] This section does discuss the `std::printf()` and `std::scanf()` families of functions, 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()`.

How the output is formatted is controlled by the given `format` string. All of its characters 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 5-6 explains the available conversion specifiers. The expected argument types listed are for the case where no length modifier is used (discussed later).

Table 5-6. *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).

(*continued*)

[1]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 5-6. (*continued*)

Specifier	Description
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" (it assumes a using namespace 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:

```
%<flags><width><precision><length_modifier><conversion>
```

with

- `<flags>`: Zero or more flags that change the meaning of the conversion specifier. See Table 5-7.
- `<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 5-8 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.[2]
- `<conversion>`: The only required component, which specifies the conversion to apply to the argument. (See Table 5-6.)

[2] `wint_t` is defined in `<cwchar>` and is a `typedef` 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 5-7. *Available Flags*

Flag	Description
-	Left-justify the output. By default, output is right justified.
+	Always output the sign of a number, even for positive numbers.
space-character	Prefix the output with a space if the number to output is non-negative or results in no characters. Ignored if + is also specified.
#	Output 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.

Table 5-8. *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

The modifiers in Table 5-8 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`.

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 5-9. The last column shows the argument type in case no length modifiers are specified (see Table 5-10).

Table 5-9. *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.	`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*`

(*continued*)

Table 5-9. (*continued*)

Specifier	Matches...	Argument
`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 non-whitespace characters.	`char**`
`[...]`	A non-empty 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 non-whitespace 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:

```
%<*><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 5-10. 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 5-9.

Table 5-10. *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 non-obvious difference between Table 5-10 and Table 5-8 is that by default, floating-point arguments must point to a `float` and not a `double`.

Example

```
std::string s = "int: +123; double: -2.34E-3; chars: abcdef";
int i = 0; double d = 0.0; char chars[4] = { 0 };
std::sscanf(s.data(), "int: %i; double: %lE; chars: %[abc]", &i, &d, chars);
std::printf("int: %+i; double: %.2lE; chars: %s", i, d, chars);
```

CHAPTER 6

Characters and Strings

Strings

<string>

The Standard 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.

To illustrate, we will mostly use `std::string`. Everything in this section, though, applies equally well to all types. The locale and regular expression functionalities discussed thereafter 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 well, except for the emplacement functions (which are of little use for characters).

P. Van Weert and M. Gregoire, *C++ Standard Library Quick Reference*,
DOI 10.1007/978-1-4842-1876-1_6

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 (that is, 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 four 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
```

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, or a character buffer of which the first `n` values are used. 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.

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`.

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,] 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 its `vector` counterpart, 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::make_pair(3u, L"c is a pair<unsigned, wstring>"s); // <pair>
```

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 (==, <, >=, and so on), or their compare() member. The latter has the following overloads:

```
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 or a negative/positive number if the first string is less/greater than the second.

String Conversions

To parse various types of integral numbers from a string, a series of non-member functions of the following form has been defined:

```
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. If an index pointer is provided, it receives the index of the first character that is not converted.

Similarly, to parse floating-point numbers, a set of functions exists of the following form:

```
float stof(const (w)string&, size_t* index = nullptr);
```

stof(), stod(), and stold() are provided to convert to float, double, and long double, respectively.

To perform the opposite conversion and convert from numerical types to a (w)string, functions to_(w)string(*X*) are provided, where *X* can be int, unsigned, long, unsigned long, long long, unsigned long long, float, double, or long double. The returned value is a std::(w)string.

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) (defined only for ints that represent chars) and std::isw*class*(wint_t) (analogous; wint_t is an integral typedef), where *class* equals one of the values in Table 6-1. All functions return a non-zero 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', and so on.
print	Printable characters: digits, letters, space, punctuation marks, and so on.
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 the integral int and wint_t types. If the conversion is not defined or possible, these functions simply return their input value.

The exact behavior of all character classifications and transformations 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, you also see that the C++ `<locale>` header offers a list of overloads for `std::is`*`class`*`()` and `std::tolower()` / `toupper()` (all templated on the character type) that use a given `locale` rather than the active `C` locale.

Character-Encoding Conversion `<locale>, <codecvt>`

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:

Method	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:

```
typedef std::codecvt<char32_t,char,std::mbstate_t> cvt;
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 (similar tricks can be applied to make any protected function publically accessible, not just a destructor):

```
// Forward all constructors and make protected destructor public:
template <class Base> class deletable : public Base {
  public:
    template <typename... Params>
    deletable(Params&&... params) : Base(std::forward<Params>(params)...)
    {}
    ~deletable() {}
};
```

To make the code compile, you then replace the first line with the following[2]:

```
typedef deletable<std::codecvt<char32_t,char,std::mbstate_t>> cvt;
```

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):

```
typedef deletable<std::codecvt_byname<char32_t,char,std::mbstate_t>> cvt;
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

[2]This example does not work in Visual Studio 2015. It compiles after replacing `char32_t` with `__int32` and `u32string` with `basic_string<__int32>`, but the result is wrong.

the wrapped buffer are called `rdbuf()`, and the current conversion state may be obtained using `state()`. 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 (that is, 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, and so on), 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, for instance, 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 character '.'
```

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. Afterwards, 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 (that is, 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 category 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; that is: `'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, for example, 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 form "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 a single time format specifier of the same grammar with optional modifier `'char format, char 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 `print`able ASCII characters except `$`, `` ` ``, and `@`) the relation `widen(narrow(c,0)) == 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&)`. Their implementations all have the following form (the return type is either `bool` or `CharT`):

```
template <typename CharT> ... function(CharT c, const std::locale& l) {
   return std::use_facet<std::ctype<CharT>>(l).function(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 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 lexicographically,

using the classic locale, and using a French locale (the locale name to use is platform specific). In addition to <locale>, it needs <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("C"));
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 (using namespace std is assumed):

```
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;
```

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):

[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.

```
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`, and so on). 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 Table 6-5 and Table 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 (that is, `\r`, `\r\n`, or `\n`).
- `<(.+)>(.*)</\1>` matches a 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).

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 digitis). 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:]]`, and so on.
[`^`*class*]	A character not of a given *class* (see main text). E.g.: `[^0-9]`, `[^[:s:]]`, and so on.
`\d`	A decimal digit character (short for `[[:d:]]` or `[[:digit:]]`).
`\s`	A whitespace character (short for `[[:s:]]` or `[[:space:]]`).
`\w`	A word character, that is: an alphanumeric or underscore character (short for `[[:w:]]` or `[_[:alnum:]]`).
`\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:]]` and so on).
`\`*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 sub-expression*, 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 sub-expression is *unmarked*, meaning the sub-match 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 sub-expression with index *integer* did earlier. Sub-expressions 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).

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 end of the target (unless `match_not_eol` is specified), or the position of a line-terminator character.
\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).
\B	Not a word boundary: both the previous and next character are either word or non-word 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*.

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.

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. The `match_xxx` flags mentioned in Table 6-6 are optionally passed to the matching functions or iterators discussed later.

[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: that is, an underscore or any alphabetic or numerical digit character.

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 `chars` 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, and so on), 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`, and so on).

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*, and so on.
`icase`	Character matches are done in a case-insensitive manner.
`nosubs`	No submatches against sub-expressions 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 adapters 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 sub-expressions 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`	Use default matching behavior (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 (for example, 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.

(*continued*)

Match Flag	Effect
`match_continuous`	The pattern only matches sequences that start at the beginning of the target sequence (implied for `regex_match()`).
`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 sub-expression 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 sub-expression did not participate in the match. The latter occurs, for example, if the sub-expression 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
<<	Nonmember 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 sub-expressions) 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 (non-inclusive). 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 (non-inclusive) 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 `typedefs` 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 non-trivial 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 constructor creates 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 "Bold", "b", "bold again", and then "b". When omitted, this argument defaults to 0, indicating only full pattern sub_matches are to be enumerated (the example then prints "Bold" and "bold again").

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 example be used to split a comma-separated string into its different fields (or tokens). The regular expression used in that case is simply ",".

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 sub-expression 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`	Use default formatting (this constant has value zero).
`format_sed`	Use the same syntax as the POSIX utility `sed` 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.

CHAPTER 7

Concurrency

Threads <thread>

Threads are the basic building blocks to be able to write code that runs in parallel.

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 equivalent:

```
std::thread worker1(function, "arg", anotherArg);
std::thread worker2([=] { function("arg", anotherArg); });
```

The function with its arguments is called in a newly launched thread of execution prior to returning from the `thread`'s constructor.

Both the function and its arguments must first be copied or moved (for example, 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: for example, by wrapping it using `std::ref()` / `std::cref()`. Of course, you can also simply use a lambda expression with capture-by-reference. Functors, reference wrappers, and lambda expressions are all discussed in detail in Chapter 2.

The `thread` class does not offer any facilities to retrieve the function'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 threads for longer-running concurrent tasks that do not necessarily return a result.

P. Van Weert and M. Gregoire, *C++ Standard Library Quick Reference*,
DOI 10.1007/978-1-4842-1876-1_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 function 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 function 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 function has finished executing
- `detach()`: Disassociates the `thread` object from the possibly continuing thread of execution

Note that detaching a `thread` is the only standard way to asynchronously execute a function in a fire-and-forget manner.

A `std::thread` offers no means to terminate, interrupt, or resume the underlying thread of execution. Stopping the thread function 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 (for example, for logging purposes).
- They can be compared using == (for example, 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 (`<`, `>=`, and so on) 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 also 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.

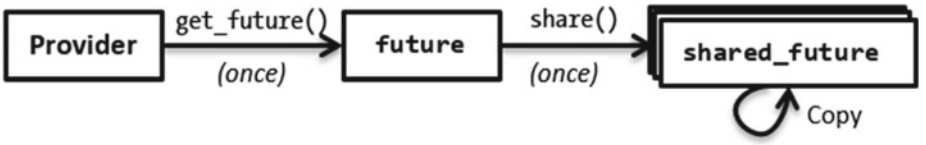
To wait until the result is ready without actually retrieving it, use one of the wait functions: wait(), wait_until(time_point), or wait_for(duration). The former waits indefinitely, and the latter two wait no longer than a timeout specified using one of the types defined in <chrono> (Chapter 2).

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:

```
int gcd(int x, int y) { return y? gcd(y, x % y) : x; } // Euclid's alg.
```

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` (that is, 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 (that is, 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);
std::future<int> gcd_future = gcd_task.get_future();
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 an `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 re-executing 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.

Asides from a large selection of mutex and lock types, the `<mutex>` header also defines `std::call_once()`, which is used to ensure that a given function is called only once. The `call_once()` utility is introduced at the end of this section.

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. The simplest, leanest lock is `lock_guard`, which simply calls `lock()` in its constructor and `unlock()` in its destructor:

```
{  std::lock_guard<std::mutex> 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::lock_guard<std::mutex> 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 `lock_guard` 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_timed_mutex`	No	Yes	Yes	`<shared_mutex>`
`shared_mutex`[1]	No	No	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`.[2]

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 non-recursive 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.

---

[1] Scheduled to be added by the C++17 version of the Standard Library.

[2] Although normally uncommon, `try_lock()` is allowed to spuriously fail: that is, return `false` even though the mutex is not owned by any other thread. Take that into account when designing more advanced synchronization scenarios.

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` 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

There are three lock types provided by the `standard: std::lock_guard`, `unique_lock`, and `shared_lock`.

std::lock_guard

`lock_guard` is a trivial, textbook RAII-style template class: by default, it locks a mutex in its constructor and unlocks it in its destructor. The only additional member is a constructor intended to be used with a mutex already owned by the calling thread. This constructor is called by passing the global `std::adopt_lock` constant:

```
std::lock_guard<std::mutex> lock(m, std::adopt_lock);
```

std::unique_lock

Although `lock_guard` is easy and optimally efficient, it is limited in functionality. To facilitate more advanced scenarios, the standard defines `unique_lock`.

The basic usage is the same:

```
std::unique_lock<std::mutex> lock(m);
```

However, unique_lock has several additional features compared to a lock_guard, including these:

- A unique_lock can be moved and swapped (but of course not copied).
- It has a release() function to disassociate it from the underlying mutex without unlocking it.
- The mutex() member returns a pointer to the underlying mutex.

What really sets unique_lock apart, though, is that it offers functions to release and (re)acquire ownership of the 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. The locking functions of unique_lock may be called only once, even if the underlying mutex is recursive, or an exception will be thrown. To check whether the unique_lock will unlock upon destruction, call owns_lock() (unique_lock also casts to a Boolean with this value).

In addition to the obvious constructor with a given mutex, the unique_lock class supports three alternative constructors where you pass an additional constant:

- adopt_lock: Used when the mutex is already owned by the current thread (analogous to the equivalent lock_guard constructor).
- 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>

Both lock_guard and unique_lock 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. Even though they acquire shared ownership, the names of its locking and unlocking members do not contain shared. This is done to ensure that a shared_lock satisfies the requirements for other utilities such as std::lock() and std::condition_variable_any, both discussed later.

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. The Standard Library offers templated helper functions instead to facilitate this:

```
std::lock(lockable1, lockable2, ..., lockableN);
```

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_` or `shared_locks` (for example, constructed with `defer_lock`). Although the standard does not specify how this should be achieved, if all threads use `std::lock()`, there are no deadlocks.

Of course, 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 0-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, 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. Deadlock detection is only optional, though: 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");
```

Only a single thread that calls `call_once()` with a given instance of `std::once_flag`—a default-constructible, non-copyable, non-moveable 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<std::mutex> 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>`.[3] 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; });
```

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.

---

[3]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`.

Notification normally occurs because the condition has changed:

```
{  std::lock_guard<std::mutex> 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.[4]

There is one more notification function, but it is a nonmember 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();
```

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 non-memory-related resource limitation.

Destructing a condition variable upon which a thread is still waiting results in undefined behavior.

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.

[4]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.

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, injects 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 intra-thread 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 (for example, 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 generally 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 Typedefs

The `atomic<T>` template may be used at least with any trivially copyable[5] type `T`, and specializations are defined for Booleans as well as all other integral types and pointer types `T*`. The latter two offer additional operations, as described later.

For the Boolean and integral specializations, convenience `typedef`s 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 `typedef` abbreviates the first words of the `xxx` type:

typedef	xxx	typedef	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`		

[5] 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 bit-wise copied (for example, using `memcpy()`).

Common 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. A constructor to initialize with a given `T` value is present as well. This initialization is not atomic, though: concurrent access from another thread, even through atomic operations, is a data race. Atomic variables cannot be copied, moved, or swapped.

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 as regular `T` variables:

```
std::atomic_int atom;        // Uninitialized!
atom = 123;
std::cout << atom << std::endl;        // 123
```

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.

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& oldVal, T newVal) {
  if (load() == oldVal) {
      store(newVal); return true;
  } else {
      oldVal = load(); return false;
}}
```

Naturally, though, both 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.

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 in the beginning of a singly linked list:

```
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;
```

All operations introduced in this section are atomic for any base type T. For types such as Booleans, integers, and pointers, most compilers simply generate a few special instructions that guarantee atomicity (most current CPUs support this). If so, lock_free() returns true. For other types, atomic variables mostly resort to mutex-like constructs to accomplish atomicity. For such types, lock_free() returns false.

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!
```

Atomic Operations for Integral and Pointer Types

Certain template specializations offer additional operators that atomically update the variable. The selection is based on which atomic instructions current hardware generally supports (no multiplication, for example):

- 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(), and so on) contain release fences that synchronize with the acquire fences in operations that

read from the same variable (`load()`, exchanges, `fetch_xxx()`, and so forth). 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 (the name strings for example), 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 at all times. 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 whether the default constructor initializes the flag. The only 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 other members:

- `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.

Nonmember Functions

For compatibility with C, `<atomic>` defines nonmember counterparts 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 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 restrict 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 this:

```
#ifdef NDEBUG
 #define assert(_)
#else
 #define assert(CONDITION) if (!CONDITION) { print_msg(...); 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. 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, or function pre- and postconditions. 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
```

Caution Make sure 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 is not evaluated if `NDEBUG` is defined (for example, for a release build).

P. Van Weert and M. Gregoire, *C++ Standard Library Quick Reference*,
DOI 10.1007/978-1-4842-1876-1_8

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.

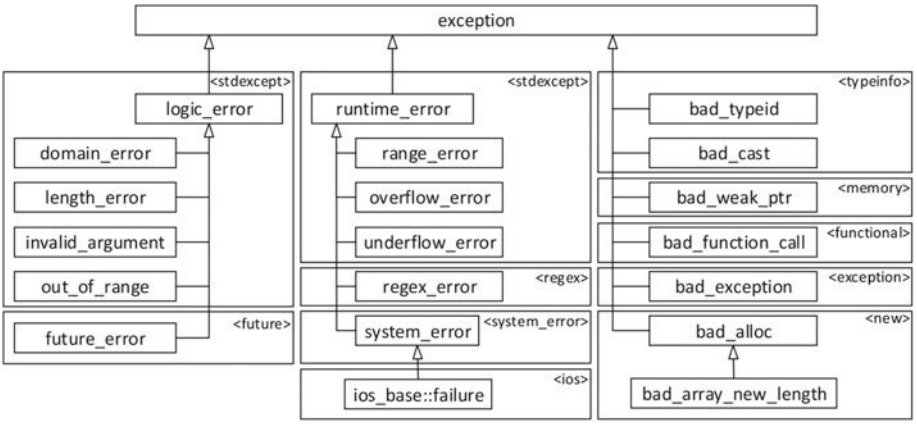


Figure 8-1. *The C++ Standard Library exception hierarchy*

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 (Unicode strings encoded as UTF-8 could be used, for instance; see 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_errors` represent avoidable errors in the program's logic, whereas `runtime_errors` are caused by less predictable events beyond the scope of the program. `logic_error`, `runtime_error`, and most of their subclasses (except `system_errors` and `future_error`, which require an error code, as discussed later) must be passed a `std::string` or `const char*` pointer upon construction, which is returned by `what()` afterward. There is thus no need to further override `what()`.

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. An `exception_ptr` can point to a value of any type, not just an `std::exception`. It can 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 (that is, a reference-counted smart pointer may be used to implement `exception_ptr`).

A couple of functions are 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 (remember, this can have any type) currently in flight when called from inside a `catch()` block, either directly or indirectly (a `catch()` block may call, for example, 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 `t`.

`[[noreturn]] void std::rethrow_exception(exception_ptr)`

> 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_ptr`s 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 that you not use nested_exception directly, but use these helper methods instead:

```
[[noreturn]] template<typename T> 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. Being a std::nested_exception as well, 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, 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);
   }
}
```

[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.

```
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 (see 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: Error codes and conditions belong to a category. The category singleton objects are responsible for converting between both numberings.
- system_error: An exception class (see Figure 8-1) with an extra code() member 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>

In addition 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.

All this may seem fairly complicated, but the main uses of these errors remain straightforward. To compare a given error code, such as that from a caught `system_error` exception, with either an error condition or a code, the == and != operators can 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()` member returning an `error_code` that can be compared with known code values (see 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, non-copyable 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:

Member	Description
`name()`	Returns the category's name (as a `const char*`).
`message()`	Returns an explanatory `std::string` for a given error condition value (an `int`).
`default_error_condition()`	Converts a given error code value (an `int`) to a portable `error_condition`.
`equivalent()`	Compares error codes with portable conditions. It is 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 that sets 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 set of constructors and methods similar to `error_code`, except

- It does not have a `default_error_condition()` method or equivalent function to go from error condition to error code.
- Error *condition* enumerations are used instead of error *code* enumerations: 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 (std::string-to-numeric conversions, for example). 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!
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 it can be something like the following:

```
Error: result out of range
```

For completeness, we show two alternative ways of reporting an error string for the current errno. They use, respectively, strerror() from <cstring> (take care: this function is not thread-safe!) and std::perror() from <cstdio>. The following two lines print a message similar to the earlier code:

```
std::cerr << "Error: " << std::strerror(errno) << std::endl;
std::perror("Error");     // Prefix string is non-optional
```

Failure Handling <exception>

std::uncaught_exception()

If, anywhere in your code, you want to know whether an exception is currently in flight that has not been caught yet—in other words, detect that stack unwinding is in progress—use uncaught_exception(), which returns true if that is the case.

[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.

■ **Note** There usually is no reason or safe way to use `uncaught_exception()`, so we advise against using it. It is just mentioned here for completeness.

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 *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 the bug that triggered the process to `terminate()`. You should consider setting this up for any professional application.

std::unexpected()

The runtime calls `std::unexpected()` if a dynamic exception specification[3] is disregarded: that is, if a function throws something it is not allowed to. Analogous to `terminate()`, this function calls an `std::unexpected_handler` function that can be managed using `std::set_unexpected()` / `get_unexpected()`. The default handler calls `std::terminate()`.

■ **Note** Both dynamic exception specifications and `std::unexpected()` have been deprecated and are only mentioned here for completeness.

[3] A dynamic exception specification is part of the function declaration and specifies, with a comma-separated list, which exceptions that function is allowed to throw. For example: `ReturnType Func(...) throw(exception1, exception2, ...);`.

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
protocol_not_supported	EPROTONOSUPPORT

(*continued*)

Table 8-2. (*continued*)

std::errc enum Value	<cerrno> Macro
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 A

Standard Library Headers

The C++ Standard Library consists of 79 header files, of which 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. 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, and more.
<cstdint>	A set of typedefs for integral types with certain width requirements: for example, int32_t and int_fast64_t.
<limits>	numeric_limits, offering properties—such as min(), max(), lowest(), infinity(), quiet_NaN(), and so on—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, and so on. 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.
<ccomplex>	*Simply includes* <complex>.
<ctgmath>	*Includes* <cmath> *and* <ccomplex>.

(*continued*)

P. Van Weert and M. Gregoire, *C++ Standard Library Quick Reference*,
DOI 10.1007/978-1-4842-1876-1

Header	Contents
<ratio>	The ratio template, helper templates for performing arithmetic operations and comparisons on them, and a set of predefined ratios.
<random>	Pseudo-random 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 and piecewise_construct. Functions make_pair(), swap(), forward(), move(), move_if_noexcept(), and *declval()*.
<tuple>	tuple, helper classes tuple_size and tuple_element, and functions make_tuple(), forward_as_tuple(), tie(), tuple_cat(), and get().
<memory>	Smart pointers: unique_ptr, shared_ptr, and weak_ptr. *Default allocators.*
<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), functor negators, std::function, bind(), and mem_fn().
<initializer_list>	The definition of initializer_list.
<chrono>	Time utilities: durations, time_points, and clocks (steady_clock, system_clock, and high_resolution_clock).
<ctime>	C-style time and date utilities such as the tm struct, time(), localtime(), and strftime().
<cstdio>	C-style file utilities: remove(), rename(), tmpfile(), and tmpnam(). Also provides C-style stream I/O functionality: see Chapter 5.
<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. Chapter 4 discusses input/output iterators and the predefined iterator adaptors: reverse_iterator, move_iterator, and insert iterators. Stream iterators are discussed in Chapter 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.

Algorithms (Chapter 4)

Header	Contents
<algorithm>	All available algorithms, except those that are in <numeric>.
<numeric>	Numerical algorithms: accumulate(), adjacent_difference(), inner_product(), partial_sum(), and iota().

Stream I/O (Chapter 5)

Header	Contents
<ios>	ios_base, basic_ios, and fpos, typedefs ios and wios, and types streamoff, streampos, wstreampos, and streamsize. Non-parametric I/O manipulators such as boolalpha, dec, scientific, and so on.
<iomanip>	Parametric I/O manipulators such as setbase(), setfill(), get_money(), put_time(), and more.
<ostream>	basic_ostream, and typedefs ostream and wostream. The endl, ends, and flush output manipulators.
<istream>	basic_istream and basic_iostream, and typedefs istream, wistream, iostream, and wiostream. The ws input manipulator.
<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 typedefs.
<fstream>	File streams: basic_ifstream, basic_ofstream, basic_fstream, and basic_filebuf, and related typedefs.
<streambuf>	basic_streambuf, and typedefs streambuf and wstreambuf.
<iosfwd>	Forward declarations for all stream I/O types.
<cstdio>	The C-style I/O library. Basic file utilities (see Chapter 2), *plus fopen(), fclose(), and so on.* Functions for formatted (printf(), scanf(), and so on) *and character-based I/O (getc(), putc(), and so on*). It is generally recommended that you 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.*

Characters and Strings (Chapter 6)

Header	Contents
<string>	basic_string, and typedefs string, wstring, u16string, and u32string. Conversion functions such as stoi(), stof(), to_string(), and so on.
<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(), and so on.*
<cctype>	Functions to classify and transform characters: isdigit(), isspace(), tolower(), toupper(), and so on.
<cwctype>	Wide character versions of functions from <cctype>: iswdigit(), iswspace(), towlower(), towupper(), and so on.
<codecvt>	Unicode character encoding conversion facets: codecvt_utf8, codecvt_utf16, and codecvt_utf8_utf16.
<*cuchar*>	*Functions to convert between 16 or 32-bit character and multibyte sequences: c16rtomb(), c32rtomb(), mbrtoc16(), and 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, and so on.
<clocale>	lconv and the setlocale() and localeconv() functions. setlocale() only changes the C locale.
<regex>	Everything related to regular expressions.

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, 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`, and so on. 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++.*
`<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 that you use type-safe variadic templates instead.*
`<cstdbool>`	*The `__bool_true_false_are_defined` macro: expands to 1 for C++.*
`<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()`, and so on. 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

A

B

P. Van Weert and M. Gregoire, *C++ Standard Library Quick Reference*,
DOI 10.1007/978-1-4842-1876-1

C

D

E

F

G

H

I, J

K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y, Z