Finders

A *finder* is a concept that determines a position in a range corresponding to some specified criteria, usually a predicate or a regular expression. Boost String Algorithms provides some generators for producing finders in the `<boost/algorithm/string/finder.hpp>` header.

For example, the `nth_finder` generator accepts a range `r` and an index `n`, and it creates a finder that will search a range (taken as a `begin` and an `end` iterator) for the `n`th occurrence of `r`, as Listing 15-30 illustrates.

```
#include <boost/algorithm/string/finder.hpp>

TEST_CASE("boost::algorithm::nth_finder finds the nth occurrence") {
  const auto finder = boost::algorithm::nth_finder("na", 1); ❶
  std::string name("Carl Brutananadilewski"); ❷
  const auto result = finder(name.begin(), name.end()); ❸
  REQUIRE(result.begin() == name.begin() + 12); ❹ // Brutana(n)adilewski
  REQUIRE(result.end() == name.begin() + 14); ❺ // Brutanana(d)ilewski
}
```

Listing 15-30: The `nth_finder` generator creates a finder that locates the `n`th occurrence of a sequence.

You use the `nth_finder` generator to create `finder`, which will locate the second instance of `na` in a range (`n` is zero based) ❶. Next, you construct `name` containing `Carl Brutananadilewski` ❷ and invoke `finder` with the `begin` and `end` iterators of `name` ❸. The `result` is a range whose `begin` points to the second *n* in `Brutana`**`n`**`adilewski` ❹ and whose `end` points to the first *d* in `Brutanana`**`d`**`ilewski` ❺.

Table 15-14 lists the finders available in `<boost/algorithm/string/finder.hpp>`. In this table, `s` is a string, `p` is an element comparison predicate, `n` is an integral value, `beg` and `end` are iterators, `rgx` is a regular expression, and `r` is a string range.

Table 15-14: Finders in the Boost String Algorithms Library

Generator	Creates a finder that, when invoked, returns . . .
`first_finder(s, p)`	The first element matching **s** using **p**
`last_finder(s, p)`	The last element matching **s** using **p**
`nth_finder(s, p, n)`	The **n**th element matching **s** using **p**
`head_finder(n)`	The first **n** elements
`tail_finder(n)`	the last **n** elements
`token_finder(p)`	The character matching **p**
`range_finder(r)` `range_finder(beg, end)`	**r** regardless of input
`regex_finder(rgx)`	The first substring matching **rgx**

NOTE *Boost String Algorithms specifies a formatter concept, which presents the results of a finder to a replace algorithm. Only an advanced user will need these algorithms. Refer to the documentation for the* `find_format` *algorithms in the* `<boost/algorithm/string/find_format.hpp>` *header for more information.*

Modifying Algorithms

Boost contains a *lot* of algorithms for modifying a `string` (range). Between the `<boost/algorithm/string/case_conv.hpp>`, `<boost/algorithm/string/trim.hpp>`, and `<boost/algorithm/string/replace.hpp>` headers, algorithms exist to convert case, trim, replace, and erase many different ways.

For example, the `to_upper` function will convert all of a string's letters to uppercase. If you want to keep the original unmodified, you can use the `to_upper_copy` function, which will return a new object. Listing 15-31 illustrates `to_upper` and `to_upper_copy`.

```
#include <boost/algorithm/string/case_conv.hpp>

TEST_CASE("boost::algorithm::to_upper") {
  std::string powers("difficulty controlling the volume of my voice"); ❶
  SECTION("upper-cases a string") {
    boost::algorithm::to_upper(powers); ❷
    REQUIRE(powers == "DIFFICULTY CONTROLLING THE VOLUME OF MY VOICE"); ❸
  }
  SECTION("_copy leaves the original unmodified") {
    auto result = boost::algorithm::to_upper_copy(powers); ❹
    REQUIRE(powers == "difficulty controlling the volume of my voice"); ❺
    REQUIRE(result == "DIFFICULTY CONTROLLING THE VOLUME OF MY VOICE"); ❻
  }
}
```

Listing 15-31: Both `to_upper` and `to_upper_copy` convert the case of a `string`.

You create a `string` called `powers` ❶. The first test invokes `to_upper` on `powers` ❷, which modifies it in place to contain all uppercase letters ❸. The second test uses the `_copy` variant to create a new `string` called `result` ❹. The `powers` string is unaffected ❺, whereas `result` contains an all uppercase version ❻.

Some Boost String Algorithms, such as `replace_first`, also have case-insensitive versions. Just prepend an `i`, and matching will proceed regardless of case. For algorithms like `replace_first` that also have `_copy` variants, any permutation will work (`replace_first`, `ireplace_first`, `replace_first_copy`, and `ireplace_first_copy`).

The `replace_first` algorithm and its variants accept an input range `s`, a match range `m`, and a replace range `r`, and replaces the first instance of `m` in `s` with `r`. Listing 15-32 illustrates `replace_first` and `i_replace_first`.

```
#include <boost/algorithm/string/replace.hpp>

TEST_CASE("boost::algorithm::replace_first") {
  using namespace boost::algorithm;
  std::string publisher("No Starch Press"); ❶
  SECTION("replaces the first occurrence of a string") {
    replace_first(publisher, "No", "Medium"); ❷
    REQUIRE(publisher == "Medium Starch Press"); ❸
  }
  SECTION("has a case-insensitive variant") {
    auto result = ireplace_first_copy(publisher, "NO", "MEDIUM"); ❹
    REQUIRE(publisher == "No Starch Press"); ❺
    REQUIRE(result == "MEDIUM Starch Press"); ❻
  }}
```

Listing 15-32: Both `replace_first` and `i_replace_first` replace matching `string` sequences.

Here, you construct a `string` called `publisher` containing `No Starch Press` ❶. The first test invokes `replace_first` with `publisher` as the input string, `No` as the match string, and `Medium` as the replacement string ❷. Afterward, `publisher` contains `Medium Starch Press` ❸. The second test uses the `ireplace_first_copy` variant, which is case insensitive and performs a copy. You pass `NO` and `MEDIUM` as the match and replace strings ❹, respectively, and the `result` contains `MEDIUM Starch Press` ❻, whereas `publisher` is unaffected ❺.

Table 15-15 lists many of the modifying algorithms available in Boost String Algorithms. In this table, `r`, `s`, `s1`, and `s2` are strings; `p` is an element comparison predicate; `n` is an integral value; and `rgx` is a regular expression.

Table 15-15: Modifying Algorithms in the Boost String Algorithms Library

Algorithm	Description
`to_upper(`**`s`**`)` `to_upper_copy(`**`s`**`)`	Converts **`s`** to all uppercase
`to_lower(`**`s`**`)` `to_lower_copy(`**`s`**`)`	Converts **`s`** to all lowercase

`trim_left_copy_if(`**`s`**`, [`**`p`**`])`
`trim_left_if(`**`s`**`, [`**`p`**`])`
`trim_left_copy(`**`s`**`)`
`trim_left(`**`s`**`)`
	Removes leading spaces from **`s`**

`trim_right_copy_if(`**`s`**`, [`**`p`**`])`
`trim_right_if(`**`s`**`, [`**`p`**`])`
`trim_right_copy(`**`s`**`)`
`trim_right(`**`s`**`)`
	Removes trailing spaces from **`s`**

`trim_copy_if(`**`s`**`, [`**`p`**`])`
`trim_if(`**`s`**`, [`**`p`**`])`
`trim_copy(`**`s`**`)`
`trim(`**`s`**`)`
	Removes leading and trailing spaces from **`s`**

`replace_first(`**`s1, s2, r`**`)`
`replace_first_copy(`**`s1, s2, r`**`)`
`ireplace_first(`**`s1, s2, r`**`)`
`ireplace_first_copy(`**`s1, s2, r`**`)`
	Replaces the first occurrence of **`s2`** in **`s1`** with **`r`**

Algorithm	Description

`erase_first(s1, s2)`
`erase_first_copy(s1, s2)`
`ierase_first(s1, s2)`
`ierase_first_copy(s1, s2)`
	Erases the first occurrence of **s2** in **s1**

`replace_last(s1, s2, r)`
`replace_last_copy(s1, s2, r)`
`ireplace_last(s1, s2, r)`
`ireplace_last_copy(s1, s2, r)`
	Replaces the last occurrence of **s2** in **s1** with **r**

`erase_last(s1, s2)`
`erase_last_copy(s1, s2)`
`ierase_last(s1, s2)`
`ierase_last_copy(s1, s2)`
	Erases the last occurrence of **s2** in **s1**

`replace_nth(s1, s2, n, r)`
`replace_nth_copy(s1, s2, n, r)`
`ireplace_nth(s1, s2, n, r)`
`ireplace_nth_copy(s1, s2, n, r)`
	Replaces the **n**th occurrence of **s2** in **s1** with **r**

`erase_nth(s1, s2, n)`
`erase_nth_copy(s1, s2, n)`
`ierase_nth(s1, s2, n)`
`ierase_nth_copy(s1, s2, n)`
	Erases the **n**th occurrence of **s2** in **s1**

`replace_all(s1, s2, r)`
`replace_all_copy(s1, s2, r)`
`ireplace_all(s1, s2, r)`
`ireplace_all_copy(s1, s2, r)`
	Replaces all occurrences of **s2** in **s1** with **r**

`erase_all(s1, s2)`
`erase_all_copy(s1, s2)`
`ierase_all(s1, s2)`
`ierase_all_copy(s1, s2)`
	Erases all occurrences of **s2** in **s1**

`replace_head(s, n, r)` `replace_head_copy(s, n, r)`	Replaces the first **n** characters of **s** with **r**
`erase_head(s, n)` `erase_head_copy(s, n)`	Erases the first **n** characters of **s**
`replace_tail(s, n, r)` `replace_tail_copy(s, n, r)`	Replaces the last **n** characters of **s** with **r**
`erase_tail(s, n)` `erase_tail_copy(s, n)`	Erases the last **n** characters of **s**
`replace_regex(s, rgx, r)` `replace_regex_copy(s, rgx, r)`	Replaces the first instance of **rgx** in **s** with **r**
`erase_regex(s, rgx)` `erase_regex_copy(s, rgx)`	Erases the first instance of **rgx** in **s**
`replace_all_regex(s, rgx, r)` `replace_all_regex_copy(s, rgx, r)`	Replaces all instances of **rgx** in **s** with **r**
`erase_all_regex(s, rgx)` `erase_all_regex_copy(s, rgx)`	Erases all instances of **rgx** in **s**

Splitting and Joining

Boost String Algorithms contains functions for splitting and joining strings in the `<boost/algorithm/string/split.hpp>` and `<boost/algorithm/string/join.hpp>` headers.

To split a string, you provide the split function with an STL container res, a range s, and a predicate p. It will tokenize the range s using the predicate p to determine delimiters and insert the results into res. Listing 15-33 illustrates the split function.

```
#include <vector>
#include <boost/algorithm/string/split.hpp>
#include <boost/algorithm/string/classification.hpp>

TEST_CASE("boost::algorithm::split splits a range based on a predicate") {
  using namespace boost::algorithm;
  std::string publisher("No Starch Press"); ❶
  std::vector<std::string> tokens; ❷
  split(tokens, publisher, is_space()); ❸
  REQUIRE(tokens[0] == "No"); ❹
  REQUIRE(tokens[1] == "Starch");
  REQUIRE(tokens[2] == "Press");
}
```

Listing 15-33: The split function tokenizes a string.

Armed again with publisher ❶, you create a vector called tokens to contain the results ❷. You invoke split with tokens as the results container, publisher as the range, and an is_space as your predicate ❸. This splits the publisher into pieces by spaces. Afterward, tokens contains No, Starch, and Press as expected ❹.

You can perform the inverse operation with join, which accepts an STL container seq and a separator string sep. The join function will bind each element of seq together with sep between each.

Listing 15-34 illustrates the utility of join and the indispensability of the Oxford comma.

```
#include <vector>
#include <boost/algorithm/string/join.hpp>

TEST_CASE("boost::algorithm::join staples tokens together") {
  std::vector<std::string> tokens{ "We invited the strippers",
                                   "JFK", "and Stalin." }; ❶
  auto result = boost::algorithm::join(tokens, ", "); ❷
  REQUIRE(result == "We invited the strippers, JFK, and Stalin."); ❸
}
```

Listing 15-34: The join function attaches string tokens together with a separator.

You instantiate a vector called tokens with three string objects ❶. Next, you use join to bind token's constituent elements together with a comma followed by a space ❷. The result is a single string containing the constituent elements bound together with commas and spaces ❸.

Table 15-16 lists many of the split/join algorithms available in `<boost/algorithm/string/split.hpp>` and `<boost/algorithm/string/join.hpp>`. In this table, `res`, `s`, `s1`, `s2`, and `sep` are strings; `seq` is a range of strings; `p` is an element comparison predicate; and `rgx` is a regular expression.

Table 15-16: `split` and `join` Algorithms in the Boost String Algorithms Library

Function	Description

`find_all(res, s1, s2)`
`ifind_all(res, s1, s2)`
`find_all_regex(res, s1, rgx)`
`iter_find(res, s1, s2)`
	Finds all instances of `s2` or `rgx` in `s1`, writing each into `res`

`split(res, s, p)`
`split_regex(res, s, rgx)`
`iter_split(res, s, s2)`
	Split `s` using `p`, `rgx`, or `s2`, writing tokens into `res`

`join(seq, sep)`	Returns a `string` joining `seq` using `sep` as a separator
`join_if(seq, sep, p)`	Returns a `string` joining all elements of `seq` matching `p` using `sep` as a separator

Searching

Boost String Algorithms offers a handful of functions for searching ranges in the `<boost/algorithm/string/find.hpp>` header. These are essentially convenient wrappers around the finders in Table 15-8.

For example, the `find_head` function accepts a range `s` and a length `n`, and it returns a range containing the first `n` elements of `s`. Listing 15-35 illustrates the `find_head` function.

```
#include <boost/algorithm/string/find.hpp>

TEST_CASE("boost::algorithm::find_head computes the head") {
  std::string word("blandishment"); ❶
  const auto result = boost::algorithm::find_head(word, 5); ❷
  REQUIRE(result.begin() == word.begin()); ❸ // (b)landishment
  REQUIRE(result.end() == word.begin()+5); ❹ // bland(i)shment
}
```

Listing 15-35: The `find_head` function creates a range from the beginning of a `string`.

You construct a `string` called `word` containing `blandishment` ❶. You pass it into `find_head` along with the length argument `5` ❷. The `begin` of `result` points to the beginning of `word` ❸, and its `end` points to 1 past the fifth element ❹.

Table 15-17 lists many of the find algorithms available in `<boost/algorithm/string/find.hpp>`. In this table, `s`, `s1`, and `s2` are strings; `p` is an element comparison predicate; `rgx` is a regular expression; and `n` is an integral value.

Table 15-17: Find Algorithms in the Boost String Algorithms Library

Predicate	Finds the . . .
`find_first(s1, s2)` `ifind_first(s1, s2)`	First instance of `s2` in `s1`
`find_last(s1, s2)` `ifind_last(s1, s2)`	First instance of `s2` in `s1`
`find_nth(s1, s2, n)` `ifind_nth(s1, s2, n)`	`n`th instance of `s2` in `s1`
`find_head(s, n)`	First `n` characters of `s`
`find_tail(s, n)`	Last `n` characters of `s`
`find_token(s, p)`	First character matching `p` in `s`
`find_regex(s, rgx)`	First substring matching `rgx` in `s`
`find(s, fnd)`	Result of applying `fnd` to `s`

Boost Tokenizer

Boost Tokenizer's `boost::tokenizer` is a class template that provides a view of a series of tokens contained in a `string`. A `tokenizer` takes three optional template parameters: a tokenizer function, an iterator type, and a string type.

The *tokenizer function* is a predicate that determines whether a character is a delimiter (returns `true`) or not (returns `false`). The default tokenizer function interprets spaces and punctuation marks as separators. If you want to specify the delimiters explicitly, you can use the `boost::char_separator<char>` class, which accepts a C-string containing all the delimiting characters. For example, a `boost::char_separator<char>(";|,")` would separate on semicolons (`;`), pipes (`|`), and commas (`,`).

The iterator type and string type correspond with the type of `string` you want to split. By default, these are `std::string::const_iterator` and `std::string`, respectively.

Because `tokenizer` doesn't allocate memory and `boost::algorithm::split` does, you should strongly consider using the former whenever you only need to iterate over the tokens of a `string` once.

A `tokenizer` exposes `begin` and `end` methods that return input iterators, so you can treat it as a range of values corresponding to the underlying token sequence.

Listing 15-36 tokenizes the iconic palindrome `A man, a plan, a canal, Panama!` by comma.

```
#include<boost/tokenizer.hpp>
#include<string>

TEST_CASE("boost::tokenizer splits token-delimited strings") {
  std::string palindrome("A man, a plan, a canal, Panama!"); ❶
  boost::char_separator<char> comma{ "," }; ❷
  boost::tokenizer<boost::char_separator<char>> tokens{ palindrome, comma }; ❸
  auto itr = tokens.begin(); ❹
```

```
  REQUIRE(*itr == "A man"); ❺
  itr++; ❻
  REQUIRE(*itr == " a plan");
  itr++;
  REQUIRE(*itr == " a canal");
  itr++;
  REQUIRE(*itr == " Panama!");
}
```

Listing 15-36: The `boost::tokenizer` splits strings by specified delimiters.

Here, you construct `palindrome` ❶, `char_separator` ❷, and the corresponding `tokenizer` ❸. Next, you extract an iterator from the tokenizer using its `begin` method ❹. You can treat the resulting iterator as usual, dereferencing its value ❺ and incrementing to the next element ❻.

Localizations

A *locale* is a class for encoding cultural preferences. The locale concept is typically encoded in whatever operating environment your application runs within. It also controls many preferences, such as string comparison; date and time, money, and numeric formatting; postal and ZIP codes; and phone numbers.

The STL offers the `std::locale` class and many helper functions and classes in the `<locale>` header.

Mainly for brevity (and partially because English speakers are the primary intended audience for this book), this chapter won't explore locales any further.

Summary

This chapter covered `std::string` and its ecosystem in detail. After exploring its similarities to `std::vector`, you learned about its built-in methods for handling human-language data, such as comparing, adding, removing, replacing, and searching. You looked at how the numeric conversion functions allow you to convert between numbers and strings, and you examined the role that `std::string_view` plays in passing strings around your programs. You also learned how to employ regular expressions to perform intricate match, search, and replacement based on potentially complicated patterns. Finally, you trekked through the Boost String Algorithms library, which complements and extends the built-in methods of `std::string` with additional methods for searching, replacing, trimming, erasing, splitting, and joining.

EXERCISES

15-1. Refactor the histogram calculator in Listings 9-30 and 9-31 to use `std::string`. Construct a `string` from the program's input and modify `AlphaHistogram` to accept a `string_view` or a `const string&` in its `ingest` method. Use a range-based `for` loop to iterate over the ingested elements of `string`. Replace the `counts` field's type with an associative container.

15-2. Implement a program that determines whether the user's input is a palindrome.

15-3. Implement a program that counts the number of vowels in the user's input.

15-4. Implement a calculator program that supports addition, subtraction, multiplication, and division of any two numbers. Consider using the `find` method of `std::string` and the numeric conversion functions.

15-5. Extend your calculator program in some of the following ways: permit multiple operations or the modulo operator and accept floating-point numbers or parentheses.

15-6. Optional: Read more about locales in [localization].

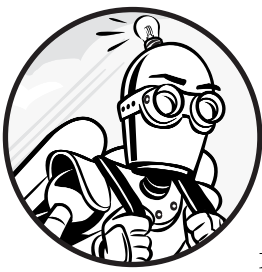
FURTHER READING

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

16

STREAMS

Either write something worth reading or
do something worth writing.
—Benjamin Franklin



This chapter introduces streams, the major concept that enables you to connect inputs from any kind of source and outputs to any kind of destination using a common framework. You'll learn about the classes that form the base elements of this common framework, several built-in facilities, and how to incorporate streams into user-defined types.

Streams

A *stream* models a *stream of data*. In a stream, data flows between objects, and those objects can perform arbitrary processing on the data. When you're working with streams, output is data going into the stream and input is data coming out of the stream. These terms reflect the streams as viewed from the user's perspective.

In C++, streams are the primary mechanism for performing input and output (I/O). Regardless of the source or destination, you can use streams as the common language to connect inputs to outputs. The STL uses class inheritance to encode the relationships between various stream types. The primary types in this hierarchy are:

- The `std::basic_ostream` class template in the `<ostream>` header that represents an output device
- The `std::basic_istream` class template in the `<istream>` header that represents an input device
- The `std::basic_iostream` class template in the `<iostream>` header for devices that are input and output

All three stream types require two template parameters. The first corresponds to the stream's underlying data type and the second to a traits type.

This section covers streams from a user's perspective rather than from a library implementer's perspective. You'll understand the streams interface and know how to interact with standard I/O, files, and strings using the STL's built-in stream support. If you must implement a new kind of stream (for example, for a new library or framework), you'll need a copy of the ISO C++ 17 Standard, some working examples, and an ample supply of coffee. I/O is complicated, and you'll see this difficulty reflected in a stream implementation's internal complexity. Fortunately, a well-designed stream class hides much of this complexity from users.

Stream Classes

All STL stream classes that users interact with derive from `basic_istream`, `basic_ostream`, or both via `basic_iostream`. The headers that declare each type also provide `char` and `wchar_t` specializations for those templates, as outlined in Table 16-1. These heavily used specializations are particularly useful when you're working with human-language data input and output.

Table 16-1: Template Specializations for the Primary Stream Templates

Template	Parameter	Specialization	Header
`basic_istream`	`char`	`istream`	`<istream>`
`basic_ostream`	`char`	`ostream`	`<ostream>`
`basic_iostream`	`char`	`iostream`	`<iostream>`
`basic_istream`	`wchar_t`	`wistream`	`<istream>`
`basic_ostream`	`wchar_t`	`wostream`	`<ostream>`
`basic_iostream`	`wchar_t`	`wiostream`	`<iostream>`

The objects in Table 16-1 are abstractions that you can use in your programs to write generic code. Do you want to write a function that logs output to an arbitrary source? If so, you can accept an `ostream` reference parameter and not deal with all the nasty implementation details. (Later in the "Output File Streams" on page 542, you'll learn how to do this.)

Often, you'll want to perform I/O with the user (or the program's environment). Global stream objects provide a convenient, stream-based wrapper for you to work against.

Global Stream Objects

The STL provides several *global stream objects* in the `<iostream>` header that wrap the input, output, and error streams stdin, stdout, and stderr. These implementation-defined standard streams are preconnected channels between your program and its executing environment. For example, in a desktop environment, stdin typically binds to the keyboard and stdout and stderr bind to the console.

NOTE *Recall that in Part I you saw extensive use of* `printf` *to write to stdout.*

Table 16-2 lists the global stream objects, all of which reside in the `std` namespace.

Table 16-2: The Global Stream Objects

Object	Type	Purpose
`cout` `wcout`	`ostream` `wostream`	Output, like a screen
`cin` `wcin`	`istream` `wistream`	Input, like a keyboard
`cerr` `wcerr`	`ostream` `wostream`	Error output (unbuffered)
`clog` `wclog`	`ostream` `wostream`	Error output (buffered)

So how do you use these objects? Well, stream classes support operations that you can partition into two categories:

Formatted operations Might perform some preprocessing on their input parameters before performing I/O

Unformatted operations Perform I/O directly

The following sections explain each of these categories in turn.

Formatted Operations

All formatted I/O passes through two functions: the *standard stream operators*, `operator<<` and `operator>>`. You'll recognize these as the left and right shift operators from "Logical Operators" on page 182. Somewhat confusingly, streams overload the left and right shift operators with completely unrelated functionality. The semantic meaning of the expression `i << 5` depends entirely on the type of `i`. If `i` is an integral type, this expression means *take* *i and shift the bits to the left by five binary digits*. If `i` is not an integral type, it means *write the value 5 into* `i`. Although this notational collision is unfortunate, in practice it doesn't cause too much trouble. Just pay attention to the types you're using and test your code well.

Output streams overload `operator<<`, which is referred to as the *output operator* or the *inserter*. The `basic_ostream` class template overloads the output operator for all fundamental types (except `void` and `nullptr_t`) and some STL containers, such as `basic_string`, `complex`, and `bitset`. As an `ostream` user, you need not worry about how these overloads translate objects into readable output.

Listing 16-1 illustrates how to use the output operator to write various types into `cout`.

```
#include <iostream>
#include <string>
#include <bitset>

using namespace std;

int main() {
  bitset<8> s{ "01110011" };
  string str("Crying zeros and I'm hearing ");
  size_t num{ 111 };
  cout << s; ❶
  cout << '\n'; ❷
  cout << str; ❸
  cout << num; ❹
  cout << "s\n"; ❺
}
--------------------------------------------------------------------------------
01110011 ❶❷
Crying zeros and I'm hearing 111s ❸❹❺
```

Listing 16-1: Using cout and operator<< to write into stdout

You use the output `operator<<` to write a `bitset` ❶, a `char` ❷, a `string` ❸, a `size_t` ❹, and a null-terminated string literal ❺ to stdout via `cout`. Even though you write five distinct types to the console, you never deal with serialization issues. (Consider the hoops you would have had to jump through to get `printf` to yield similar output given these types.)

One very nice feature of the standard stream operators is that they generally return a reference to the stream. Conceptually, overloads are typically defined along the following lines:

```
ostream& operator<<(ostream&, char);
```

This means you can chain output operators together. Using this technique, you can refactor Listing 16-1 so `cout` appears only once, as Listing 16-2 illustrates.

```
#include <iostream>
#include <string>
#include <bitset>

using namespace std;

int main() {
  bitset<8> s{ "01110011" };
  string str("Crying zeros and I'm hearing ");
  size_t num{ 111 };
  cout << s << '\n' << str << num << "s\n"; ❶
}
--------------------------------------------------------------------------------
01110011
Crying zeros and I'm hearing 111s ❶
```

Listing 16-2: Refactoring Listing 16-1 by chaining output operators together

Because each invocation of `operator<<` returns a reference to the output stream (here, `cout`), you simply chain the calls together to obtain identical output ❶.

Input streams overload `operator>>`, which is referred to as the *input operator* or the *extractor*. The `basic_istream` class has corresponding overloads for the input operator for all the same types as `basic_ostream`, and again as a user, you can largely ignore the deserialization details.

Listing 16-3 illustrates how to use the input operator to read two `double` objects and a `string` from `cin`, then print the implied mathematical operation's result to stdout.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  double x, y;
  cout << "X: ";
  cin >> x; ❶
  cout << "Y: ";
  cin >> y; ❷

  string op;
  cout << "Operation: ";
  cin >> op; ❸
  if (op == "+") {
    cout << x + y; ❹
  } else if (op == "-") {
    cout << x - y; ❺
  } else if (op == "*") {
    cout << x * y; ❻
  } else if (op == "/") {
```

```
    cout << x / y; ❼
  } else {
    cout << "Unknown operation " << op; ❽
  }
}
```

Listing 16-3: A primitive calculator program using `cin` and `operator<<` to collect input

Here, you collect two `doubles` `x` ❶ and `y` ❷ followed by the `string` `op` ❸, which encodes the desired operation. Using an `if` statement, you can output the specified operation's result for addition ❹, subtraction ❺, multiplication ❻, and division ❼, or indicate to the user that `op` is unknown ❽.

To use the program, you type the requested values into the console when directed. A newline will send the input (as stdin) to `cin`, as Listing 16-4 illustrates.

```
X: 3959 ❶
Y: 6.283185 ❷
Operation: * ❸
24875.1 ❹
```

Listing 16-4: A sample run of the program in Listing 16-3 that calculates the circumference of Earth in miles

You input the two `double` objects: the radius of Earth in miles, **`3959`** ❶ and 2π, **`6.283185`** ❷, and you specify multiplication **`*`** ❸. The result is Earth's circumference in miles ❹. Note that you don't need to provide a decimal point for an integral value ❶; the stream is smart enough to know that there's an implicit decimal.

NOTE *You might wonder what happens in Listing 16-4 if you input a non-numeric string for `X` ❶ or `Y` ❷. The stream enters an error state, which you'll learn about later in this chapter in the "Stream State" section on page 530. In an error state, the stream ceases to accept input, and the program won't accept any more input.*

Unformatted Operations

When you're working with text-based streams, you'll usually want to use formatted operators; however, if you're working with binary data or if you're writing code that needs low-level access to streams, you'll want to know about the unformatted operations. Unformatted I/O involves a lot of detail. For brevity, this section provides a summary of the relevant methods, so if you need to use unformatted operations, refer to [input.output].

The `istream` class has many unformatted input methods. These methods manipulate streams at the byte level and are summarized in Table 16-3. In this table, `is` is of type `std::istream <T>`, `s` is a `char*`, `n` is a stream size, `pos` is a position type, and `d` is a delimiter of type `T`.

Table 16-3: Unformatted Read Operations for `istream`

Method	Description
`is.get([c])`	Returns next character or writes to character reference **c** if provided.
`is.get(s, n, [d])` `is.getline(s, n, [d])`	The operation `get` reads up to **n** characters into the buffer **s**, stopping if it encounters a newline, or **d** if provided. The operation `getline` is the same except it reads the newline character as well. Both write a terminating null character to **s**. You must ensure **s** has enough space.
`is.read(s, n)` `is.readsome(s, n)`	The operation `read` reads up to **n** characters into the buffer **s**; encountering end of file is an error. The operation `readsome` is the same except it doesn't consider end of file an error.
`is.gcount()`	Returns the number of characters read by **is**'s last unformatted read operation.
`is.ignore()`	Extracts and discards a single character.
`is.ignore(n, [d])`	Extracts and discards up to **n** characters. If **d** is provided, `ignore` stops if **d** is found.
`is.peek()`	Returns the next character to be read without extracting.
`is.unget()`	Puts the last extracted character back into the string.
`is.putback(c)`	If **c** is the last character extracted, executes `unget`. Otherwise, sets the `badbit`. Explained in the "Stream State" section.

Output streams have corollary unformatted write operations, which manipulate streams at a very low level, as summarized in Table 16-4. In this table, `os` is of type `std::ostream <T>`, `s` is a `char*`, and `n` is a stream size.

Table 16-4: Unformatted Write Operations for `ostream`

Method	Description
`os.put(c)`	Writes **c** to the stream
`os.write(s, n)`	Writes **n** characters from **s** to the stream
`os.flush()`	Writes all buffered data to the underlying device

Special Formatting for Fundamental Types

All fundamental types, in addition to `void` and `nullptr`, have input and output operator overloads, but some have special rules:

`char` and `wchar_t` The input operator skips whitespace when assigning character types.

`char*` and `wchar_t*` The input operator first skips whitespace and then reads the string until it encounters another whitespace or an end-of-file (EOF). You must reserve enough space for the input.

`void*` Address formats are implementation dependent for input and output operators. On desktop systems, addresses take hexadecimal literal form, such as `0x01234567` for 32-bit or `0x0123456789abcdef` for 64-bit.

`bool` The input and output operators treat Boolean values as numbers: 1 for `true` and 0 for `false`.

Numeric types The input operator requires that input begin with at least one digit. Badly formed input numbers yield a zero-valued result.

These rules might seem a bit strange at first, but they're fairly straightforward once you get used to them.

NOTE *Avoid reading into C-style strings, because it's up to you to ensure that you've allocated enough space for the input data. Failure to perform adequate checking results in undefined behavior and possibly major security vulnerabilities. Use* `std::string` *instead.*

Stream State

A stream's state indicates whether I/O failed. Each stream type exposes the constant static members referred to collectively as its *bits*, which indicate a possible stream state: `goodbit`, `badbit`, `eofbit`, and `failbit`. To determine whether a stream is in a particular state, you invoke member functions that return a `bool` indicating whether the stream is in the corresponding state. Table 16-5 lists these member functions, the stream state corresponding to a `true` result, and the state's meaning.

Table 16-5: The Possible Stream States, Their Accessor Methods, and Their Meanings

Method	State	Meaning
`good()`	`goodbit`	The stream is in a good working state.
`eof()`	`eofbit`	The stream encountered an EOF.
`fail()`	`failbit`	An input or output operation failed, but the stream might still be in a good working state.
`bad()`	`badbit`	A catastrophic error occurred, and the stream is not in a good state.

NOTE *To reset a stream's status to indicate a good working state, you can invoke its* `clear()` *method.*

Streams implement an implicit bool conversion (`operator bool`), so you can check whether a stream is in a good working state simply and directly. For example, you can read input from stdin word by word until it encounters an EOF (or some other failure condition) using a simple `while` loop. Listing 16-5 illustrates a simple program that uses this technique to generate word counts from stdin.

```
#include <iostream>
#include <string>
```

```
int main() {
  std::string word; ❶
  size_t count{}; ❷
  while (std::cin >> word) ❸
    count++; ❹
  std::cout << "Discovered " << count << " words.\n"; ❺
}
```

Listing 16-5: A program that counts words from stdin

You declare a string called word to receive words from stdin ❶, and you initialize a count variable to zero ❷. Within the while loop's Boolean expression, you attempt to assign new input into word ❸. When this succeeds, you increment count ❹. Once it fails—for example, due to encountering an EOF—you cease incrementing and print the final tally ❺.

You can try two methods to test Listing 16-5. First, you can simply invoke the program, enter some input, and provide an EOF. How to send EOF depends on your operating system. In the Windows command line, you can enter EOF by pressing CTRL-Z and pressing enter. In Linux bash or in the OS X shell, you press CTRL-D. Listing 16-6 demonstrates how to invoke Listing 16-5 from the Windows command line.

```
$ listing_16_5.exe ❶
Size matters not. Look at me. Judge me by my size, do you? Hmm? Hmm. And well
you should not. For my ally is the Force, and a powerful ally it is. Life
creates it, makes it grow. Its energy surrounds us and binds us. Luminous
beings are we, not this crude matter. You must feel the Force around you;
here, between you, me, the tree, the rock, everywhere, yes. ❷
^Z ❸
Discovered 70 words. ❹
```

Listing 16-6: Invoking the program in Listing 16-5 by typing input into the console

First, you invoke your program ❶. Next, enter some arbitrary text followed by a new line ❷. Then issue EOF. The Windows command line shows the somewhat cryptic sequence **^Z** on the command line, after which you must press ENTER. This causes std::cin to enter the eofbit state, ending the while loop in Listing 16-5 ❸. The program indicates that you've sent 70 words into stdin ❹.

On Linux and Mac and in Windows PowerShell, you have another option. Rather than entering the input directly into the console, you can save the text to a file, say *yoda.txt*. The trick is to use cat to read the text file and then use the pipe operator | to send the contents to your program. The pipe operator "pipes" the stdout of the program to its left into the stdin of the program on the right. The following command illustrates this process:

```
$ cat yoda.txt❶ |❷ ./listing_15_4❸
Discovered 70 words.
```

The cat command reads the contents of *yoda.txt* ❶. The pipe operator ❷ pipes the stdout of cat into stdin of listing_15_4 ❸. Because cat sends EOF when it encounters the end of *yoda.txt*, you don't need to enter it manually.

Sometimes you'll want streams to throw an exception when certain fail bits occur. You can do this easily with a stream's exceptions method, which accepts a single argument corresponding to the bit you want to throw exceptions. If you desire multiple bits, you can simply join them together using Boolean OR (|).

Listing 16-7 illustrates how to refactor Listing 16-5 so it handles the badbit with exceptions and eofbit/failbit with the default handling.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  cin.exceptions(istream::badbit); ❶
  string word;
  size_t count{};
  try { ❷
    while(cin >> word) ❸
      count++;
    cout << "Discovered " << count << " words.\n"; ❹
  } catch (const std::exception& e) { ❺
    cerr << "Error occurred reading from stdin: " << e.what(); ❻
  }
}
```

Listing 16-7: Refactoring Listing 16-5 to handle badbit with exceptions

You start the program by invoking the exceptions method on std::cin ❶. Because cin is an istream, you pass istream::badbit as the argument of exception, indicating that you want cin to throw an exception any time it gets into a catastrophic state. To account for possible exceptions, you wrap the existing code in a try-catch block ❷, so if cin sets badbit while it's reading input ❸, the user never receives a message about the word count ❹. Instead, the program catches the resulting exception ❺ and prints the error message ❻.

Buffering and Flushing

Many ostream class templates involve operating system calls under the hood, for example, to write to a console, a file, or a network socket. Relative to other function calls, system calls are usually slow. Rather than invoking a system call for each output element, an application can wait for multiple elements and then send them all together to improve performance.

The queuing behavior is called *buffering*. When the stream empties the buffered output, it's called *flushing*. Usually, this behavior is completely transparent to the user, but sometimes you want to manually flush the ostream. For this (and other tasks), you turn to manipulators.

Manipulators

Manipulators are special objects that modify how streams interpret input or format output. Manipulators exist to perform many kinds of stream alterations. For example, `std::ws` modifies an `istream` to skip over whitespace. Here are some other manipulators that work on `ostreams`:

- `std::flush` empties any buffered output directly to an `ostream`.
- `std::ends` sends a null byte.
- `std::endl` is like `std::flush` except it sends a newline before flushing.

Table 16-6 summarizes the manipulators in the `<istream>` and `<ostream>` headers.

Table 16-6: Four Manipulators in the `<istream>` and `<ostream>` Headers

Manipulator	Class	Behavior
`ws`	`istream`	Skips over all whitespaces
`flush`	`ostream`	Writes any buffered data to the stream by invoking its `flush` method
`ends`	`ostream`	Sends a null byte
`endl`	`ostream`	Sends a newline and flushes

For example, you could replace ❹ in Listing 16-7 with the following:

```
cout << "Discovered " << count << " words." << endl;
```

This will print a newline and also flush output.

NOTE *As a general rule, use* `std::endl` *when your program has finished outputting text to the stream for a while and* `\n` *when you know your program will output more text soon.*

The stdlib provides many other manipulators in the `<ios>` header. You can, for example, determine whether an `ostream` will represent Boolean values textually (`boolalpha`) or numerically (`noboolalpha`); integral values as octal (`oct`), decimal (`dec`), or hexadecimal (`hex`); and floating-point numbers as decimal notation (`fixed`) or scientific notation (`scientific`). Simply pass one of these manipulators to an `ostream` using `operator<<` and *all* subsequent insertions of the corresponding type will be manipulated (not just an immediately preceding operand).

You can also set a stream's width parameter using the `setw` manipulator. A stream's width parameter has varied effects, depending on the stream. For example, with `std::cout`, `setw` will fix the number of output characters allocated to the next output object. Additionally, for floating-point output, `setprecision` will set the following numbers' precision.

Listing 16-8 illustrates how these manipulators perform functions similar to those of the various `printf` format specifiers.

```
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
  cout << "Gotham needs its " << boolalpha << true << " hero."; ❶
  cout << "\nMark it " << noboolalpha << false << "!"; ❷
  cout << "\nThere are " << 69 << "," << oct << 105 << " leaves in here."; ❸
  cout << "\nYabba " << hex << 3669732608 << "!"; ❹
  cout << "\nAvogadro's number: " << scientific << 6.0221415e-23; ❺
  cout << "\nthe Hogwarts platform: " << fixed << setprecision(2) << 9.750123; ❻
  cout << "\nAlways eliminate " << 3735929054; ❼
  cout << setw(4) << "\n"
       << 0x1 << "\n"
       << 0x10 << "\n"
       << 0x100 << "\n"
       << 0x1000 << endl; ❽
}
--------------------------------------------------------------------------------
Gotham needs its true hero. ❶
Mark it 0! ❷
There are 69,151 leaves in here. ❸
Yabba dabbad00! ❹
Avogadro's Number: 6.022142e-23 ❺
the Hogwarts platform: 9.75 ❻
Always eliminate deadc0de ❼
1
10
100
1000 ❽
```

Listing 16-8: A program illustrating some of the manipulators available in the `<iomanip>` header

The `boolalpha` manipulator in the first line causes Boolean values to print textually as `true` and `false` ❶, whereas `noboolalpha` causes them to print as 1 and 0 instead ❷. For integral values, you can print as octal with `oct` ❸ or hexadecimal with `hex` ❹. For floating-point values, you can specify scientific notation with `scientific` ❺, and you can set the number of digits to print with `setprecision` and specify decimal notation with `fixed` ❻. Because manipulators apply to all subsequent objects you insert into a stream, when you print another integral value at the end of the program, the last integral manipulator (`hex`) applies, so you get a hexadecimal representation ❼. Finally, you employ `setw` to set the field width for output to 4, and you print some integral values ❽.

Table 16-7 summarizes this sampling of common manipulators.

Table 16-7: Many of the Manipulators Available in the `<iomanip>` Header

Manipulator	Behavior
`boolalpha`	Represents Booleans textually rather than numerically.
`noboolalpha`	Represents Booleans numerically rather than textually.
`oct`	Represents integral values as octal.
`dec`	Represents integral values as decimal.
`hex`	Represents integral values as hexadecimal.
`setw(`**`n`**`)`	Sets the width parameter of a stream to **`n`**. The exact effect depends on the stream.
`setprecision(`**`p`**`)`	Specifies floating-point precision as **`p`**.
`fixed`	Represents floating-point numbers in decimal notation.
`scientific`	Represents floating-point numbers in scientific notation.

NOTE *Refer to Chapter 15 in* The C++ Standard Library, *2nd Edition, by Nicolai M. Josuttis or [iostream.format].*

User-Defined Types

You can make user-defined types work with streams by implementing certain non-member functions. To implement the output operator for type `YourType`, the following function declaration serves most purposes:

```
ostream&❶ operator<<(ostream&❷ s, const YourType& m ❸);
```

For most cases, you'll simply return ❶ the same `ostream` you receive ❷. It's up to you how to send output into the `ostream`. But typically, this involves accessing fields on `YourType` ❸, optionally performing some formatting and transformations, and then using the output operator. For example, Listing 16-9 shows how to implement an output operator for `std::vector` to print its size, capacity, and elements.

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename T>
ostream& operator<<(ostream& s, vector<T> v) { ❶
  s << "Size: " << v.size()
    << "\nCapacity: " << v.capacity()
    << "\nElements:\n"; ❷
  for (const auto& element : v)
    s << "\t" << element << "\n"; ❸
  return s; ❹
}
```

```
int main() {
  const vector<string> characters {
    "Bobby Shaftoe",
    "Lawrence Waterhouse",
    "Gunter Bischoff",
    "Earl Comstock"
  }; ❺
  cout << characters << endl; ❻

  const vector<bool> bits { true, false, true, false }; ❼
  cout << boolalpha << bits << endl; ❽
}
```

```
Size: 4
Capacity: 4
Elements: ❷
        Bobby Shaftoe ❸
        Lawrence Waterhouse ❸
        Gunter Bischoff ❸
        Earl Comstock ❸

Size: 4
Capacity: 32
Elements: ❷
        true ❸
        false ❸
        true ❸
        false ❸
```

Listing 16-9: A program illustrating how to implement an output operator for a vector

First, you define a custom output operator as a template, using the template parameter as the template parameter of `std::vector` ❶. This allows you to use the output operator for many kinds of `vector`s (as long as the type `T` also supports the output operator). The first three lines of output give the size and capacity of `vector`, as well as the title `Elements` indicating that the elements of the `vector` follow ❷. The following `for` loop iterates over each element in the `vector`, sending each on a separate line to the `ostream` ❸. Finally, you return the stream reference `s` ❹.

Within `main`, you initialize a `vector` called `characters` containing four `string`s ❺. Thanks to your user-defined output operator, you can simply send `characters` to `cout` as if it were a fundamental type ❻. The second example uses a `vector<bool>` called `bits`, which you also initialize with four elements ❼ and print to stdout ❽. Notice that you use the `boolalpha` manipulator, so when your user-defined output operator runs, the `bool` elements print textually ❸.

You can also provide user-defined input operators, which work similarly. A simple corollary is as follows:

```
istream&❶ operator>>(istream&❷ s, YourType& m ❸);
```

As with the output operator, the input operator typically returns ❶ the same stream it receives ❷. However, unlike with the output operator, the `YourType` reference will generally not be `const`, because you'll want to modify the corresponding object using input from the stream ❸.

Listing 16-10 illustrates how to specify an input operator for `deque` so it pushes elements into the container until an insertion fails (for example, due to an EOF character).

```
#include <iostream>
#include <deque>

using namespace std;

template <typename T>
istream& operator>>(istream& s, deque<T>& t) { ❶
  T element; ❷
  while (s >> element) ❸
    t.emplace_back(move(element)); ❹
  return s; ❺
}

int main() {
  cout << "Give me numbers: "; ❻
  deque<int> numbers;
  cin >> numbers; ❼
  int sum{};
  cout << "Cumulative sum:\n";
  for(const auto& element : numbers) {
    sum += element;
    cout << sum << "\n"; ❽
  }
}
--------------------------------------------------------------------------
Give me numbers: ❻ 1 2 3 4 5 ❼
Cumulative sum:
1  ❽
3  ❽
6  ❽
10 ❽
15 ❽
```

Listing 16-10: A program illustrating how to implement an input operator for a deque

Your user-defined input operator is a function template so you can accept any `deque` containing a type that supports the input operator ❶. First, you construct an element of type `T` so you can store input from the `istream` ❷. Next, you use the familiar `while` construct to accept input from the `istream` until the input operation fails ❸. (Recall from the "Stream State" section that streams can get into failed states in many ways, including reaching an EOF or encountering an I/O error.) After each insertion, you `move` the result into `emplace_back` on the `deque` to avoid unnecessary copies ❹. Once you're done inserting, you simply return the `istream` reference ❺.

Within `main`, you prompt the user for numbers ❻ and then use the insertion operator on a newly initialized `deque` to insert elements from stdin. In this sample program run, you input the numbers 1 to 5 ❼. For a bit of fun, you compute a cumulative sum by keeping a tally and iterating over each element, printing that iteration's result ❽.

NOTE *The preceding examples are simple user-defined implementations of input and output operators. You might want to elaborate these implementations in production code. For example, the implementations only work with `ostream` classes, which implies that they won't work with any non-`char` sequences.*

String Streams

The *string stream classes* provide facilities for reading from and writing to character sequences. These classes are useful in several situations. Input strings are especially useful if you want to parse string data into types. Because you can use the input operator, all the standard manipulator facilities are available to you. Output strings are excellent for building up strings from variable-length input.

Output String Streams

Output string streams provide output-stream semantics for character sequences, and they all derive from the class template `std::basic_ostringstream` in the `<sstream>` header, which provides the following specializations:

```
using ostringstream = basic_ostringstream<char>;
using wostringstream = basic_ostringstream<wchar_t>;
```

The output string streams support all the same features as an `ostream`. Whenever you send input to the string stream, the stream stores this input into an internal buffer. You can think of this as functionally equivalent to the `append` operation of `string` (except that string streams are potentially more efficient).

Output string streams also support the `str()` method, which has two modes of operation. Given no argument, `str` returns a copy of the internal buffer as a `basic_string` (so `ostringstream` returns a `string`; `wostringstream` returns a `wstring`). Given a single `basic_string` argument, the string stream will replace its buffer's current contents with the contents of the argument. Listing 16-11 illustrates how to use an `ostringstream`, send character data to it, build a `string`, reset its contents, and repeat.

```
#include <string>
#include <sstream>

TEST_CASE("ostringstream produces strings with str") {
  std::ostringstream ss; ❶
  ss << "By Grabthar's hammer, ";
  ss << "by the suns of Worvan. ";
```

```
  ss << "You shall be avenged."; ❷
  const auto lazarus = ss.str(); ❸

  ss.str("I am Groot."); ❹
  const auto groot = ss.str(); ❺

  REQUIRE(lazarus == "By Grabthar's hammer, by the suns"
                     " of Worvan. You shall be avenged.");
  REQUIRE(groot == "I am Groot.");
}
```

Listing 16-11: Using an `ostringstream` *to build strings*

After declaring an `ostringstream` ❶, you treat it just like any other `ostream` and use the output operator to send it three separate character sequences ❷. Next, you invoke `str` without an argument, which produces a `string` called `lazarus` ❸. Then you invoke `str` with the string literal `I am Groot` ❹, which replaces the contents of `ostringstream` ❺.

NOTE *Recall from "C-Style Strings" on page 45 that you can place multiple string literals on consecutive lines and the compiler will treat them as one. This is done purely for source code–formatting purposes.*

Input String Streams

Input string streams provide input stream semantics for character sequences, and they all derive from the class template `std::basic_istringstream` in the `<sstream>` header, which provides the following specializations:

```
using istringstream = basic_istringstream<char>;
using wistringstream = basic_istringstream<wchar_t>;
```

These are analogous to the `basic_ostringstream` specializations. You can construct input string streams by passing a `basic_string` with appropriate specialization (`string` for an `istringstream` and `wstring` for a `wistringstream`). Listing 16-12 illustrates by constructing an input string stream with a string containing three numbers and using the input operator to extract them. (Recall from "Formatted Operations" on page 525 that whitespace is the appropriate delimiter for string data.)

```
TEST_CASE("istringstream supports construction from a string") {
  std::string numbers("1 2.23606 2"); ❶
  std::istringstream ss{ numbers }; ❷
  int a;
  float b, c, d;
  ss >> a; ❸
  ss >> b; ❹
  ss >> c;
  REQUIRE(a == 1);
  REQUIRE(b == Approx(2.23606));
  REQUIRE(c == Approx(2));
```

```
  REQUIRE_FALSE(ss >> d); ❺
}
```

Listing 16-12: Using a `string` to build `istringstream` objects and extract numeric types

You construct a `string` from the literal `1 2.23606 2` ❶, which you pass into the constructor of an `istringstream` called `ss` ❷. This allows you to use the input operator to parse out `int` objects ❸ and `float` objects ❹ just like any other input stream. Once you've exhausted the stream and the output operator fails, `ss` converts to `false` ❺.

String Streams Supporting Input and Output

Additionally, if you want a string stream that supports input and output operations, you can use the `basic_stringstream`, which has the following specializations:

```
using stringstream = basic_stringstream<char>;
using wstringstream = basic_stringstream<wchar_t>;
```

This class supports the input and output operators, the `str` method, and construction from a string. Listing 16-13 illustrates how to use a combination of input and output operators to extract tokens from a string.

```
TEST_CASE("stringstream supports all string stream operations") {
  std::stringstream ss;
  ss << "Zed's DEAD"; ❶

  std::string who;
  ss >> who; ❷
  int what;
  ss >> std::hex >> what; ❸

  REQUIRE(who == "Zed's");
  REQUIRE(what == 0xdead);
}
```

Listing 16-13: Using a `stringstream` for input and output

You create a `stringstream` and sent the `Zed's DEAD` with the output operator ❶. Next, you parse `Zed's` out of the `stringstream` using the input operator ❷. Because `DEAD` is a valid hexadecimal integer, you use the input operator and the `std::hex` manipulator to extract it into an `int` ❸.

NOTE *All string streams are moveable.*

Summary of String Stream Operations

Table 16-8 provides a partial list of `basic_stringstream` operations. In this table, `ss`, `ss1`, and `ss2` are of type `std::basic_stringstream<T>`; `s` is a std::basic_string<T>; obj is a formatted object; pos is a position type; dir is a std::ios_base::seekdir; and flg is a std::ios_base::iostate.

Table 16-8: A Partial List of std::basic_stringstream Operations

Operation	Notes
basic_stringstream<**T**> { [**s**], [**om**] }	Performs braced initialization of a newly constructed string stream. Defaults to empty string **s** and in|out open mode **om**.
basic_stringstream<**T**> { move(**ss**) }	Takes ownership of **ss**'s internal buffer.
~basic_stringstream	Destructs internal buffer.
ss.rdbuf()	Returns raw string device object.
ss.str()	Gets the contents of the string device object.
ss.str(**s**)	Sets the contents of the string device object to **s**.
ss >> **obj**	Extracts formatted data from the string stream.
ss << **obj**	Inserts formatted data into the string stream.
ss.tellg()	Returns the input position index.
ss.seekg(**pos**) **ss**.seekg(**pos**, **dir**)	Sets the input position indicator.
ss.flush()	Synchronizes the underlying device.
ss.good() **ss**.eof() **ss**.bad() !**ss**	Inspects the string stream's bits.
ss.exceptions(**flg**)	Configures the string stream to throw an exception whenever a bit in **flg** gets set.
ss1.swap(**ss2**) swap(**ss1**, **ss2**)	Exchanges each element of **ss1** with those of **ss2**.

File Streams

The *file stream classes* provide facilities for reading from and writing to character sequences. The file stream class structure follows that of the string stream classes. File stream class templates are available for input, output, and both.

File stream classes provide the following major benefits over using native system calls to interact with file contents:

- You get the usual stream interfaces, which provide a rich set of features for formatting and manipulating output.
- The file stream classes are RAII wrappers around the files, meaning it's impossible to leak resources, such as files.
- File stream classes support move semantics, so you can have tight control over where files are in scope.

Opening Files with Streams

You have two options for opening a file with any file stream. The first option is the `open` method, which accepts a `const char* filename` and an optional `std::ios_base::openmode` bitmask argument. The `openmode` argument can be one of the many possible combinations of values listed in Table 16-9.

Table 16-9: Possible Stream States, Their Accessor Methods, and Their Meanings

Flag (in `std::ios`)	File	Meaning
`in`	Must exist	Read
`out`	Created if doesn't exist	Erase the file; then write
`app`	Created if doesn't exist	Append
`in|out`	Must exist	Read and write from beginning
`in|app`	Created if doesn't exist	Update at end
`out|app`	Created if doesn't exist	Append
`out|trunc`	Created if doesn't exist	Erase the file; then read and write
`in|out|app`	Created if doesn't exist	Update at end
`in|out|trunc`	Created if doesn't exist	Erase the file; then read and write

Additionally, you can add the `binary` flag to any of these combinations to put the file in *binary mode*. In binary mode, the stream won't convert special character sequences, like end of line (for example, a carriage return plus a line feed on Windows) or EOF.

The second option for specifying a file to open is to use the stream's constructor. Each file stream provides a constructor taking the same arguments as the `open` method. All file stream classes are RAII wrappers around the file handles they own, so the files will be automatically cleaned up when the file stream destructs. You can also manually invoke the `close` method, which takes no arguments. You might want to do this if you know you're done with the file but your code is written in such a way that the file stream class object won't destruct for a while.

File streams also have default constructors, which don't open any files. To check whether a file is open, invoke the `is_open` method, which takes no arguments and returns a Boolean.

Output File Streams

Output file streams provide output stream semantics for character sequences, and they all derive from the class template `std::basic_ofstream` in the `<fstream>` header, which provides the following specializations:

```
using ofstream = basic_ofstream<char>;
using wofstream = basic_ofstream<wchar_t>;
```

The default `basic_ofstream` constructor doesn't open a file, and the non-default constructor's second optional argument defaults to `ios::out`.

Whenever you send input to the file stream, the stream writes the data to the corresponding file. Listing 16-14 illustrates how to use `ofstream` to write a simple message to a text file.

```
#include <fstream>

using namespace std;

int main() {
  ofstream file{ "lunchtime.txt", ios::out|ios::app }; ❶
  file << "Time is an illusion." << endl; ❷
  file << "Lunch time, " << 2 << "x so." << endl; ❸
}
---------------------------------------------------------------------------
lunchtime.txt:
Time is an illusion. ❷
Lunch time, 2x so. ❸
```

Listing 16-14: A program opening the file lunchtime.txt *and appending a message to it. (The output corresponds to the contents of* lunchtime.txt *after a single program execution.)*

You initialize an `ofstream` called `file` with the path `lunchtime.txt` and the flags `out` and `app` ❶. Because this combination of flags appends output, any data you send through the output operator into this file stream gets appended to the end of the file. As expected, the file contains the message you passed to the output operator ❷❸.

Thanks to the `ios::app` flag, the program will append output to *lunchtime.txt* if it exists. For example, if you run the program again, you'll get the following output:

```
Time is an illusion.
Lunch time, 2x so.
Time is an illusion.
Lunch time, 2x so.
```

The second iteration of the program added the same phrase to the end of the file.

Input File Streams

Input file streams provide input stream semantics for character sequences, and they all derive from the class template `std::basic_ifstream` in the `<fstream>` header, which provides the following specializations:

```
using ifstream = basic_ifstream<char>;
using wifstream = basic_ifstream<wchar_t>;
```

The default `basic_ifstream` constructor doesn't open a file, and the non-default constructor's second optional argument defaults to `ios::in`.

Whenever you read from the file stream, the stream reads data from the corresponding file. Consider the following sample file, *numbers.txt*:

```
-54
203
9000
0
99
-789
400
```

Listing 16-15 contains a program that uses an `ifstream` to read from a text file containing integers and return the maximum. The output corresponds with invoking the program and passing the path of the file *numbers.txt*.

```
#include <iostream>
#include <fstream>
#include <limits>

using namespace std;

int main() {
  ifstream file{ "numbers.txt" }; ❶
  auto maximum = numeric_limits<int>::min(); ❷
  int value;
  while (file >> value) ❸
    maximum = maximum < value ? value : maximum; ❹
  cout << "Maximum found was " << maximum << endl; ❺
}
--------------------------------------------------------------------------
Maximum found was 9000 ❺
```

Listing 16-15: A program that reads the text file numbers.txt *and prints its maximum integer*

You first initialize an `istream` to open the *numbers.txt* text file ❶. Next, you initialize the maximum variable with the minimum value an `int` can take ❷. Using the idiomatic input stream and `while`-loop combination ❸, you cycle through each integer in the file, updating the maximum as you find higher values ❹. Once the file stream cannot parse any more integers, you print the result to stdout ❺.

Handling Failure

As with other streams, file streams fail silently. If you use a file stream constructor to open a file, you must check the `is_open` method to determine whether the stream successfully opened the file. This design differs from most other stdlib objects where invariants are enforced by exceptions. It's hard to say why the library implementors chose this approach, but the fact is that you can opt into an exception-based approach fairly easily.

You can make your own factory functions to handle file-opening failures with exceptions. Listing 16-16 illustrates how to implement an `ifstream` factory called `open`.

```
#include <fstream>
#include <string>

using namespace std;

ifstream❶ open(const char* path❷, ios_base::openmode mode = ios_base::in❸) {
  ifstream file{ path, mode }; ❹
  if(!file.is_open()) { ❺
    string err{ "Unable to open file " };
    err.append(path);
    throw runtime_error{ err }; ❻
  }
  file.exceptions(ifstream::badbit);
  return file; ❼
}
```

Listing 16-16: A factory function for generating `ifstream`*s that handle errors with exceptions rather than failing silently*

Your factory function returns an `ifstream` ❶ and accepts the same arguments as a file stream's constructor (and `open` method): a file `path` ❷ and an `openmode` ❸. You pass these two arguments into the constructor of `ifstream` ❹ and then determine whether the file opened successfully ❺. If it didn't, you throw a `runtime_error` ❻. If it did, you tell the resulting `ifstream` to throw an exception whenever its `badbit` gets set in the future ❼.

Summary of File Stream Operations

Table 16-10 provides a partial list of `basic_fstream` operations. In this table, `fs`, `fs1`, and `fs2` are of type `std:: basic_fstream <T>`; `p` is a C-style string, `std::string`, or a `std::filesystem::path`; `om` is an `std::ios_base::openmode`; `s` is a `std::basic_string<T>`; `obj` is a formatted object; `pos` is a position type; `dir` is a `std::ios_base::seekdir`; and `flg` is a `std::ios_base::iostate`.

Table 16-10: A Partial List of `std::basic_fstream` Operations

Operation	Notes
`basic_fstream<T> { [p], [om] }`	Performs braced initialization of a newly constructed file stream. If **p** is provided, attempts to open file at path **p**. Defaults to not opened and `in|out` open mode.
`basic_fstream<T> { move(fs) }`	Takes ownership of the internal buffer of **fs**.
`~basic_fstream`	Destructs internal buffer.
`fs.rdbuf()`	Returns raw string device object.
`fs.str()`	Gets the contents of the file device object.
`fs.str(s)`	Puts the contents of the file device object into **s**.

(continued)

Table 16-10: A Partial List of `std::basic_fstream` Operations (continued)

Operation	Notes
`fs >> obj`	Extracts formatted data from the file stream.
`fs << obj`	Inserts formatted data into the file stream.
`fs.tellg()`	Returns the input position index.
`fs.seekg(pos)` `fs.seekg(pos, dir)`	Sets the input position indicator.
`fs.flush()`	Synchronizes the underlying device.

`fs.good()`
`fs.eof()`
`fs.bad()`
`!fs`
	Inspects the file stream's bits.

`fs.exceptions(flg)`	Configures the file stream to throw an exception whenever a bit in `flg` gets set.
`fs1.swap(fs2)` `swap(fs1, fs2)`	Exchanges each element of `fs1` with one of `fs2`.

Stream Buffers

Streams don't read and write directly. Under the covers, they use stream buffer classes. At a high level, *stream buffer classes* are templates that send or extract characters. The implementation details aren't important unless you're planning on implementing your own stream library, but it's important to know that they exist in several contexts. The way you obtain stream buffers is by using a stream's `rdbuf` method, which all streams provide.

Writing Files to sdout

Sometimes you just want to write the contents of an input file stream directly into an output stream. To do this, you can extract the stream buffer pointer from the file stream and pass it to the output operator. For example, you can dump the contents of a file to stdout using `cout` in the following way:

```
cout << my_ifstream.rdbuf()
```

It's that easy.

Output Stream Buffer Iterators

Output stream buffer iterators are template classes that expose an output iterator interface that translates writes into output operations on the underlying stream buffer. In other words, these are adapters that allow you to use output streams as if they were output iterators.

To construct an output stream buffer iterator, use the `ostreambuf_iterator` template class in the `<iterator>` header. Its constructor takes a single output stream argument and a single template parameter corresponding to the constructor argument's template parameter (the character type). Listing 16-17 shows how to construct an output stream buffer iterator from `cout`.

```
#include <iostream>
#include <iterator>

using namespace std;

int main() {
  ostreambuf_iterator<char> itr{ cout }; ❶
  *itr = 'H'; ❷
  ++itr; ❸
  *itr = 'i'; ❹
}
---------------------------------------------------------------------------
H❷i❹
```

Listing 16-17: Writing the message `Hi` *to stdout using the* `ostreambuf_iterator` *class*

Here, you construct an output stream buffer iterator from `cout` ❶, which you write to in the usual way for an output operator: assign ❷, increment ❸, assign ❹, and so on. The result is character-by-character output to stdout. (Recall the procedures for handling output operators in "Output Iterators" on page 464.)

Input Stream Buffer Iterators

Input stream buffer iterators are template classes that expose an input iterator interface that translates reads into read operations on the underlying stream buffer. These are entirely analogous to output stream buffer iterators.

To construct an input stream buffer iterator, use the `istreambuf_iterator` template class in the `<iterator>` header. Unlike `ostreambuf_iterator`, it takes a stream buffer argument, so you must call `rdbuf()` on whichever input stream you want to adapt. This argument is optional: the default constructor of `istreambuf_iterator` corresponds to the end-of-range iterator of input iterator. For example, Listing 16-18 illustrates how to construct a string from `std::cin` using the range-based constructor of `string`.

```
#include <iostream>
#include <iterator>
#include <string>

using namespace std;

int main() {
  istreambuf_iterator<char> cin_itr{ cin.rdbuf() } ❶, end{} ❷;
  cout << "What is your name? "; ❸
  const string name{ cin_itr, end }; ❹
  cout << "\nGoodbye, " << name; ❺
}
---------------------------------------------------------------------------
What is your name? ❸josh ❹
Goodbye, josh❺
```

Listing 16-18: Constructing a string from `cin` *using input stream buffer iterators*

You construct an `istreambuf_iterator` from the stream buffer of `cin` ❶ as well as the end-of-range iterator ❷. After sending a prompt to the program's user ❸, you construct the `string name` using its range-based constructor ❹. When the user sends input (terminated by EOF), the string's constructor copies it. You then bid the user farewell using their `name` ❺. (Recall from "Stream State" on page 530 that methods for sending EOF to the console differ by operating system.)

Random Access

Sometimes you'll want random access into a stream (especially a file stream). The input and output operators clearly don't support this use case, so `basic_istream` and `basic_ostream` offer separate methods for random access. These methods keep track of the cursor or position, the index of the stream's current character. The position indicates the next byte that an input stream will read or an output stream will write.

For input streams, you can use the two methods `tellg` and `seekg`. The `tellg` method takes no arguments and returns the position. The `seekg` method allows you to set the cursor position, and it has two overloads. Your first option is to provide a `pos_type` position argument, which sets the read position. The second is to provide an `off_type` offset argument plus an `ios_base::seekdir` direction argument. The `pos_type` and `off_type` are determined by the template arguments to the `basic_istream` or `basic_ostream`, but usually these convert to/from integer types. The `seekdir` type takes one of the following three values:

- `ios_base::beg` specifies that the position argument is relative to the beginning.
- `ios_base::cur` specifies that the position argument is relative to the current position.
- `ios_base::end` specifies that the position argument is relative to the end.

For output streams, you can use the two methods `tellp` and `seekp`. These are roughly analogous to the `tellg` and `seekg` methods of input streams: the `p` stands for put and the `g` stands for get.

Consider a file *introspection.txt* with the following contents:

```
The problem with introspection is that it has no end.
```

Listing 16-19 illustrates how to employ random access methods to reset the file cursor.

```
#include <fstream>
#include <exception>
#include <iostream>

using namespace std;
```

```
ifstream open(const char* path, ios_base::openmode mode = ios_base::in) { ❶
--snip--
}

int main() {
  try {
    auto intro = open("introspection.txt"); ❷
    cout << "Contents: " << intro.rdbuf() << endl; ❸
    intro.seekg(0); ❹
    cout << "Contents after seekg(0): " << intro.rdbuf() << endl; ❺
    intro.seekg(-4, ios_base::end); ❻
    cout << "tellg() after seekg(-4, ios_base::end): "
                                                  << intro.tellg() << endl; ❼
    cout << "Contents after seekg(-4, ios_base::end): "
                                                  << intro.rdbuf() << endl; ❽
  }
  catch (const exception& e) {
    cerr << e.what();
  }
}
--------------------------------------------------------------------------------
Contents: The problem with introspection is that it has no end. ❸
Contents after seekg(0): The problem with introspection is that it has no end. ❺
tellg() after seekg(-4, ios_base::end): 49 ❼
Contents after seekg(-4, ios_base::end): end. ❽
```

Listing 16-19: A program using random access methods to read arbitrary characters in a text file

Using the factory function in Listing 16-16 ❶, you open the text file *introspection.txt* ❷. Next, you print the contents to stdout using the `rdbuf` method ❸, rewind the cursor to the first character ❹, and print the contents again. Notice that these yield identical output (because the file hasn't changed) ❺. You then use the relative offset overload of `seekg` to navigate to the fourth character from the end ❻. Using `tellg`, you learn that this is the 49th character (with zero-base indexing) ❼. When you print the input file to stdout, the output is only `end.`, because these are the last four characters in the file ❽.

NOTE *Boost offers an IOStream library with a rich set of additional features that stdlib doesn't have, including facilities for memory mapped file I/O, compression, and filtering.*

Summary

In this chapter, you learned about streams, the major concept that provides a common abstraction for performing I/O. You also learned about files as a primary source and destination for I/O. You first learned about the fundamental stream classes in the stdlib and how to perform formatted and unformatted operations, inspect stream state, and handle errors with exceptions. You learned about manipulators and how to incorporate streams into user-defined types, string streams, and file streams. This chapter culminated with stream buffer iterators, which allow you to adapt a stream to an iterator.

EXERCISES

16-1. Implement an output operator that prints information about the `AutoBrake` from "An Extended Example: Taking a Brake" on page 283. Include the vehicle's current collision threshold and speed.

16-2. Write a program that takes output from stdin, capitalizes it, and writes the result to stdout.

16-3. Read the introductory documentation for Boost IOStream.

16-4. Write a program that accepts a file path, opens the file, and prints summary information about the contents, including word count, average word length, and a histogram of the characters.

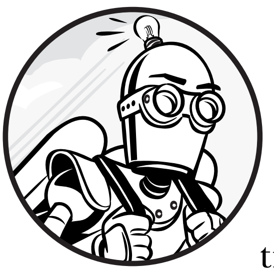
FURTHER READING

- *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference* by Angelika Langer (Addison-Wesley Professional, 2000)
- *ISO International Standard ISO/IEC (2017) — Programming Language C++* (International Organization for Standardization; Geneva, Switzerland; *https://isocpp.org/std/the-standard/*)
- *The Boost C++ Libraries*, 2nd Edition, by Boris Schäling (XML Press, 2014)

17

FILESYSTEMS

"So, you're the UNIX guru." At the time, Randy was still stupid enough to be flattered by this attention, when he should have recognized them as bone-chilling words.
—Neal Stephenson, Cryptonomicon



This chapter teaches you how to use the stdlib's Filesystem library to perform operations on filesystems, such as manipulating and inspecting files, enumerating directories, and interoperating with file streams.

The stdlib and Boost contain Filesystem libraries. The stdlib's Filesystem library grew out of Boost's, and accordingly they're largely interchangeable. This chapter focuses on the stdlib implementation. If you're interested in learning more about Boost, refer to the Boost Filesystem documentation. Boost and stdlib's implementations are mostly identical.

NOTE *The C++ Standard has a history of subsuming Boost libraries. This allows the C++ community to gain experience with new features in Boost before going through the more arduous process of including the features in the C++ Standard.*

Filesystem Concepts

Filesystems model several important concepts. The central entity is the file. A *file* is a filesystem object that supports input and output and holds data. Files exist in containers called *directories*, which can be nested within other directories. For simplicity, directories are considered files. The directory containing a file is called that file's *parent directory*.

A path is a string that identifies a specific file. Paths begin with an optional *root name*, which is an implementation-specific string, such as *C:* or *//localhost* on Windows followed by an optional root directory, which is another implementation-specific string, such as `/` on Unix-like systems. The remainder of the path is a sequence of directories separated by implementation-defined separators. Optionally, paths terminate in a non-directory file. Paths can contain the special names "`.`" and "`..`", which mean current directory and parent directory, respectively.

A *hard link* is a directory entry that assigns a name to an existing file, and a *symbolic link* (or *symlink*) assigns a name to a path (which might or might not exist). A path whose location is specified in relation to another path (usually the current directory) is called a *relative path*, and a *canonical path* unambiguously identifies a file's location, doesn't contain the special names "**.**" and "**..**", and doesn't contain any symbolic links. An *absolute path* is any path that unambiguously identifies a file's location. A major difference between a canonical path and an absolute path is that a canonical path cannot contain the special names "**.**" and "**..**".

WARNING *The stdlib filesystem might not be available if the target platform doesn't offer a hierarchical filesystem.*

std::filesystem::path

The `std::filesystem::path` is the Filesystem library's class for modeling a path, and you have many options for constructing paths. Perhaps the two most common are the default constructor, which constructs an empty path, and the constructor taking a string type, which creates the path indicated by the characters in the string. Like all other filesystem classes and functions, the `path` class resides in the `<filesystem>` header.

In this section, you'll learn how to construct a path from a `string` representation, decompose it into constituent parts, and modify it. In many common system- and application-programming contexts, you'll need to interact with files. Because each operating system has a unique representation for filesystems, the stdlib's Filesystem library is a welcome abstraction that allows you to write cross-platform code easily.

Constructing Paths

The `path` class supports comparison with other `path` objects and with `string` objects using the `operator==`. But if you just want to check whether the `path` is empty, it offers an empty method that returns a Boolean. Listing 17-1 illustrates how to construct two paths (one empty and one non-empty) and test them.

```
#include <string>
#include <filesystem>

TEST_CASE("std::filesystem::path supports == and .empty()") {
  std::filesystem::path empty_path; ❶
  std::filesystem::path shadow_path{ "/etc/shadow" }; ❷
  REQUIRE(empty_path.empty()); ❸
  REQUIRE(shadow_path == std::string{ "/etc/shadow" }); ❹
}
```

Listing 17-1: Constructing std::filesystem::path

You construct two paths: one with the default constructor ❶ and one referring to /etc/shadow ❷. Because you default construct it, the empty method of empty_path returns true ❸. The shadow_path equals a string containing /etc/shadow, because you construct it with the same contents ❹.

Decomposing Paths

The path class contains some decomposition methods that are, in effect, specialized string manipulators that allow you to extract components of the path, for example:

- root_name() returns the root name.
- root_directory() returns the root directory.
- root_path() returns the root path.
- relative_path() returns a path relative to the root.
- parent_path() returns the parent path.
- filename() returns the filename component.
- stem() returns the filename stripped of its extension.
- extension() returns the extension.

Listing 17-2 provides the values returned by each of these methods for a path pointing to a very important Windows system library, kernel32.dll.

```
#include <iostream>
#include <filesystem>

using namespace std;

int main() {
  const filesystem::path kernel32{ R"(C:\Windows\System32\kernel32.dll)" }; ❶
  cout << "Root name: " << kernel32.root_name() ❷
    << "\nRoot directory: " << kernel32.root_directory() ❸
    << "\nRoot path: " << kernel32.root_path() ❹
    << "\nRelative path: " << kernel32.relative_path() ❺
    << "\nParent path: " << kernel32.parent_path() ❻
```

```
    << "\nFilename: " << kernel32.filename() ❼
    << "\nStem: " << kernel32.stem() ❽
    << "\nExtension: " << kernel32.extension() ❾
    << endl;
}
```

```
Root name: "C:" ❷
Root directory: "\\" ❸
Root path: "C:\\" ❹
Relative path: "Windows\\System32\\kernel32.dll" ❺
Parent path: "C:\\Windows\\System32" ❻
Filename: "kernel32.dll" ❼
Stem: "kernel32" ❽
Extension: ".dll" ❾
```

Listing 17-2: A program printing various decompositions of a path

You construct a path to kernel32 using a raw string literal to avoid having to escape the backslashes ❶. You extract the root name ❷, the root directory ❸, and the root path of kernel32 ❹ and output them to stdout. Next, you extract the relative path, which displays the path relative to the root `C:\` ❺. The parent path is the path of `kernel32.dll`'s parent, which is simply the directory containing it ❻. Finally, you extract the filename ❼, its stem ❽, and its extension ❾.

Notice that you don't need to run Listing 17-2 on any particular operating system. None of the decomposition methods require that the path actually point to an existing file. You simply extract components of the path's contents, not the pointed-to file. Of course, different operating systems will yield different results, especially with respect to the delimiters (which are, for example, forward slashes on Linux).

NOTE *Listing 17-2 illustrates that `std::filesystem::path` has an `operator<<` that prints quotation marks at the beginning and end of its path. Internally, it uses `std::quoted`, a class template in the `<iomanip>` header that facilitates the insertion and extraction of quoted strings. Also, recall that you must escape the backslash in a string literal, which is why you see two rather than one in the paths embedded in the source code.*

Modifying Paths

In addition to decomposition methods, `path` offers several *modifier methods*, which allow you to modify various characteristics of a path:

- `clear()` empties the `path`.
- `make_preferred()` converts all the directory separators to the implementation-preferred directory separator. For example, on Windows this converts the generic separator / to the system-preferred separator \.
- `remove_filename()` removes the filename portion of the path.
- `replace_filename(p)` replaces the `path`'s filename with that of another `path` **p**.

- `replace_extension(p)` replaces the `path`'s extension with that of another `path` **p**.
- `remove_extension()` removes the extension portion of the path.

Listing 17-3 illustrates how to manipulate a path using several modifier methods.

```
#include <iostream>
#include <filesystem>

using namespace std;

int main() {
  filesystem::path path{ R"(C:/Windows/System32/kernel32.dll)" };
  cout << path << endl; ❶

  path.make_preferred();
  cout << path << endl; ❷

  path.replace_filename("win32kfull.sys");
  cout << path << endl; ❸

  path.remove_filename();
  cout << path << endl; ❹

  path.clear();
  cout << "Is empty: " << boolalpha << path.empty() << endl; ❺
}
```

```
"C:/Windows/System32/kernel32.dll" ❶
"C:\\Windows\\System32\\kernel32.dll" ❷
"C:\\Windows\\System32\\win32kfull.sys" ❸
"C:\\Windows\\System32\\" ❹
Is empty: true ❺
```

Listing 17-3: Manipulating a path using modifier methods. (Output is from a Windows 10 x64 system.)

As in Listing 17-2, you construct a `path` to kernel32, although this one is non-`const` because you're about to modify it ❶. Next, you convert all the directory separators to the system's preferred directory separator using `make_preferred`. Listing 17-3 shows output from a Windows 10 x64 system, so it has converted from slashes (`/`) to backslashes (`\`) ❷. Using `replace_filename`, you replace the filename from `kernel32.dll` to `win32kfull.sys` ❸. Notice again that the file described by this path doesn't need to exist on your system; you're just manipulating the path. Finally, you remove the filename using the `remove_filename` method ❹ and then empty the `path`'s contents entirely using clear ❺.

Summary of Filesystem Path Methods

Table 17-1 contains a partial listing of the available methods of `path`. Note that `p`, `p1`, and `p2` are `path` objects and `s` is a `stream` in the table.

Table 17-1: A Summary of `std::filesystem::path` Operations

Operation	Notes
`path{}`	Constructs an empty path.
`Path{ s, [f] }`	Constructs a path from the string type **s**; **f** is an optional `path::format` type that defaults to the implementation-defined pathname format.
`Path{ p }` `p1 = p2`	Copy construction/assignment.
`Path{ move(p) }` `p1 = move(p2)`	Move construction/assignment.
`p.assign(s)`	Assigns **p** to **s**, discarding current contents.
`p.append(s)` `p / s`	Appends **s** to **p**, including the appropriate separator, `path::preferred_separator`.
`p.concat(s)` `p + s`	Appends **s** to **p** without including a separator.
`p.clear()`	Erases the contents.
`p.empty()`	Returns true if **p** is empty.
`p.make_preferred()`	Converts all the directory separators to the implementation-preferred directory separator.
`p.remove_filename()`	Removes the filename portion.
`p1.replace_filename(p2)`	Replaces the filename of **p1** with that of **p2**.
`p1.replace_extension(p2)`	Replaces the extension of **p1** with that of **p2**.
`p.root_name()`	Returns the root name.
`p.root_directory()`	Returns the root directory.
`p.root_path()`	Returns the root path.
`p.relative_path()`	Returns the relative path.
`p.parent_path()`	Returns the parent path.
`p.filename()`	Returns the filename.
`p.stem()`	Returns the stem.
`p.extension()`	Returns the extension.
`p.has_root_name()`	Returns true if **p** has a root name.
`p.has_root_directory()`	Returns true if **p** has a root directory.
`p.has_root_path()`	Returns true if **p** has a root path.
`p.has_relative_path()`	Returns true if **p** has a relative path.
`p.has_parent_path()`	Returns true if **p** has a parent path.
`p.has_filename()`	Returns true if **p** has a filename.
`p.has_stem()`	Returns true if **p** has a stem.
`p.has_extension()`	Returns true if **p** has an extension.

Operation	Notes
p.c_str() p.native()	Returns the native-string representation of p.
p.begin() p.end()	Accesses the elements of a path sequentially as a half-open range.
s << p	Writes p into s.
s >> p	Reads s into p.
p1.swap(p2) swap(p1, p2)	Exchanges each element of p1 with the elements of p2.
p1 == p2 p1 != p2 p1 > p2 p1 >= p2 p1 < p2 p1 <= p2	Lexicographically compares two paths p1 and p2.

Files and Directories

The path class is the central element of the Filesystem library, but none of its methods actually interact with the filesystem. Instead, the <filesystem> header contains non-member functions to do this. Think of path objects as the way you declare which filesystem components you want to interact with and think of the <filesystem> header as containing the functions that perform work on those components.

These functions have friendly error-handling interfaces and allow you to break paths into, for example, directory name, filename, and extension. Using these functions, you have many tools for interacting with the files in your environment without having to use an operating-specific application programming interface.

Error Handling

Interacting with the environment's filesystem involves the potential for errors, such as files not found, insufficient permissions, or unsupported operations. Therefore, each non-member function in the Filesystem library that interacts with the filesystem must convey error conditions to the caller. These non-member functions provide two options: throw an exception or set an error variable.

Each function has two overloads: one that allows you to pass a reference to a std::system_error and one that omits this parameter. If you provide the reference, the function will set the system_error equal to an error condition, should one occur. If you don't provide this reference, the function will throw a std::filesystem::filesystem_error (an exception type inheriting from std::system_error) instead.

Path-Composing Functions

As an alternative to using the constructor of `path`, you can construct various kinds of paths:

- `absolute(p, [ec])` returns an absolute path referencing the same location as `p` but where `is_absolute()` is true.
- `canonical(p, [ec])` returns a canonical path referencing the same location as `p`.
- `current_path([ec])` returns the current path.
- `relative(p, [base], [ec])` returns a path where `p` is made relative to `base`.
- `temp_directory_path([ec])` returns a directory for temporary files. The result is guaranteed to be an existing directory.

Note that `current_path` supports an overload so you can set the current directory (as in `cd` or `chdir` on Posix). Simply provide a path argument, as in `current_path(p, [ec])`.

Listing 17-4 illustrates several of these functions in action.

```
#include <filesystem>
#include <iostream>

using namespace std;

int main() {
  try {
    const auto temp_path = filesystem::temp_directory_path(); ❶
    const auto relative = filesystem::relative(temp_path); ❷
    cout << boolalpha
      << "Temporary directory path: " << temp_path ❸
      << "\nTemporary directory absolute: " << temp_path.is_absolute() ❹
      << "\nCurrent path: " << filesystem::current_path() ❺
      << "\nTemporary directory's relative path: " << relative ❻
      << "\nRelative directory absolute: " << relative.is_absolute() ❼
      << "\nChanging current directory to temp.";

    filesystem::current_path(temp_path); ❽
    cout << "\nCurrent directory: " << filesystem::current_path(); ❾
  } catch(const exception& e) {
    cerr << "Error: " << e.what(); ❿
  }
}
--------------------------------------------------------------------------------
Temporary directory path: "C:\\Users\\lospi\\AppData\\Local\\Temp\\" ❸
Temporary directory absolute: true ❹
Current path: "c:\\Users\\lospi\\Desktop" ❺
Temporary directory's relative path: "..\\AppData\\Local\\Temp" ❻
Relative directory absolute: false ❼
Changing current directory to temp. ❽
Current directory: "C:\\Users\\lospi\\AppData\\Local\\Temp" ❾
```

Listing 17-4: A program using several path composing functions. (Output is from a Windows 10 x64 system.)

You construct a path using `temp_directory_path`, which returns the system's directory for temporary files ❶, and then use `relative` to determine its relative path ❷. After printing the temporary path ❸, `is_absolute` illustrates that this path is absolute ❹. Next, you print the current path ❺ and the temporary directory's path relative to the current path ❻. Because this path is relative, `is_absolute` returns `false` ❼. Once you change the path to the temporary path ❽, you then print the current directory ❾. Of course, your output will look different from the output in Listing 17-4, and you might even get an `exception` if your system doesn't support certain operations ❿. (Recall the warning at the beginning of the chapter: the C++ Standard allows that some environments might not support some or all of the filesystem library.)

Inspecting File Types

You can inspect a file's attributes given a path by using the following functions:

- `is_block_file(p, [ec])` determines if `p` is a *block file*, a special file in some operating systems (for example, block devices in Linux that allow you to transfer randomly accessible data in fixed-size blocks).
- `is_character_file(p, [ec])` determines if `p` is a *character file*, a special file in some operating systems (for example, character devices in Linux that allow you to send and receive single characters).
- `is_regular_file(p, [ec])` determines if `p` is a regular file.
- `is_symlink(p, [ec])` determines if `p` is a symlink, which is a reference to another file or directory.
- `is_empty(p, [ec])` determines if `p` is either an empty file or an empty directory.
- `is_directory(p, [ec])` determines if `p` is a directory.
- `is_fifo(p, [ec])` determines if `p` is a *named pipe*, a special kind of interprocess communication mechanism in many operating systems.
- `is_socket(p, [ec])` determines if `p` is a *socket*, another special kind of interprocess communication mechanism in many operating systems.
- `is_other(p, [ec])` determines if `p` is some kind of file other than a regular file, a directory, or a symlink.

Listing 17-5 uses `is_directory` and `is_regular_file` to inspect four different paths.

```
#include <iostream>
#include <filesystem>

using namespace std;

void describe(const filesystem::path& p) { ❶
  cout << boolalpha << "Path: " << p << endl;
  try {
    cout << "Is directory: " << filesystem::is_directory(p) << endl; ❷
```

```
    cout << "Is regular file: " << filesystem::is_regular_file(p) << endl; ❸
  } catch (const exception& e) {
    cerr << "Exception: " << e.what() << endl;
  }
}

int main() {
  filesystem::path win_path{ R"(C:/Windows/System32/kernel32.dll)" };
  describe(win_path); ❹
  win_path.remove_filename();
  describe(win_path); ❺

  filesystem::path nix_path{ R"(/bin/bash)" };
  describe(nix_path); ❻
  nix_path.remove_filename();
  describe(nix_path); ❼
}
```

Listing 17-5: A program inspecting four iconic Windows and Linux paths with `is_directory` and `is_regular_file`.

On a Windows 10 x64 machine, running the program in Listing 17-5 yielded the following output:

```
Path: "C:/Windows/System32/kernel32.dll" ❹
Is directory: false ❹
Is regular file: true ❹
Path: "C:/Windows/System32/" ❺
Is directory: true ❺
Is regular file: false ❺
Path: "/bin/bash" ❻
Is directory: false ❻
Is regular file: false ❻
Path: "/bin/" ❼
Is directory: false ❼
Is regular file: false ❼
```

And on an Ubuntu 18.04 x64 machine, running the program in Listing 17-5 yielded the following output:

```
Path: "C:/Windows/System32/kernel32.dll" ❹
Is directory: false ❹
Is regular file: false ❹
Path: "C:/Windows/System32/" ❺
Is directory: false ❺
Is regular file: false ❺
Path: "/bin/bash" ❻
Is directory: false ❻
Is regular file: true ❻
Path: "/bin/" ❼
Is directory: true ❼
Is regular file: false ❼
```

First, you define the `describe` function, which takes a single `path` ❶. After printing the path, you also print whether the path is a directory ❷ or a regular file ❸. Within `main`, you pass four different paths to `describe`:

- `C:/Windows/System32/kernel32.dll` ❹
- `C:/Windows/System32/` ❺
- `/bin/bash` ❻
- `/bin/` ❼

Note that the result is operating system specific.

Inspecting Files and Directories

You can inspect various filesystem attributes using the following functions:

- `current_path([`**`p`**`], [`**`ec`**`])`, which, if **`p`** is provided, sets the program's current path to **`p`**; otherwise, it returns the program's current path.
- `exists(`**`p`**`, [`**`ec`**`])` returns whether a file or directory exists at **`p`**.
- `equivalent(`**`p1, p2`**`, [`**`ec`**`])` returns whether **`p1`** and **`p2`** refer to the same file or directory.
- `file_size(`**`p`**`, [`**`ec`**`])` returns the size in bytes of the regular file at **`p`**.
- `hard_link_count(`**`p`**`, [`**`ec`**`])` returns the number of hard links for **`p`**.
- `last_write_time(`**`p`**`, [`**`t`**`] [`**`ec`**`])`, which, if **`t`** is provided, sets **`p`**'s last modified time to **`t`**; otherwise, it returns the last time **`p`** was modified. (**`t`** is a `std::chrono::time_point`.)
- `permissions(`**`p, prm`**`, [`**`ec`**`])` sets **`p`**'s permissions. **`prm`** is of type `std::filesystem::perms`, which is an `enum class` modeled after POSIX permission bits. (Refer to [fs.enum.perms].)
- `read_symlink(`**`p`**`, [`**`ec`**`])` returns the target of the symlink **`p`**.
- `space(`**`p`**`, [`**`ec`**`])` returns space information about the filesystem **`p`** occupies in the form of a `std::filesystem::space_info`. This POD contains three fields: `capacity` (the total size), `free` (the free space), and `available` (the free space available to a non-privileged process). All are an unsigned integer type, measured in bytes.
- `status(`**`p`**`, [`**`ec`**`])` returns the type and attributes of the file or directory **`p`** in the form of a `std::filesystem::file_status`. This class contains a `type` method that accepts no parameters and returns an object of type `std::filesystem::file_type`, which is an `enum class` that takes values describing a file's type, such as `not_found`, `regular`, `directory`. The `symlink file_status` class also offers a `permissions` method that accepts no parameters and returns an object of type `std::filesystem::perms`. (Refer to [fs.class.file_status] for details.)
- `symlink_status(`**`p`**`, [`**`ec`**`])` is like a `status` that won't follow symlinks.

If you're familiar with Unix-like operating systems, you've no doubt used the `ls` (short for "list") program many times to enumerate files and directories. On DOS-like operating systems (including Windows), you have the analogous `dir` command. You'll use several of these functions later in the chapter (in Listing 17-7) to build your own simple listing program.

Now that you know how to inspect files and directories, let's turn to how you can manipulate the files and directories your paths refer to.

Manipulating Files and Directories

Additionally, the Filesystem library contains a number of methods for manipulating files and directories:

- `copy(`**`p1, p2,`** `[`**`opt`**`], [`**`ec`**`])` copies files or directories from **`p1`** to **`p2`**. You can provide a `std::filesystem::copy_options` **`opt`** to customize the behavior of `copy_file`. This `enum class` can take several values, including `none` (report an error if the destination already exists), `skip_existing` (to keep existing), `overwrite_existing` (to overwrite), and `update_existing` (to overwrite if **`p1`** is newer). (Refer to **[**fs.enum.copy.opts**]** for details.)
- `copy_file(`**`p1, p2,`** `[`**`opt`**`], [`**`ec`**`])` is like copy except it will generate an error if **`p1`** is anything but a regular file.
- `create_directory(`**`p,`** `[`**`ec`**`])` creates the directory **`p`**.
- `create_directories(`**`p,`** `[`**`ec`**`])` is like calling `create_directory` recursively, so if a nested path contains parents that don't exist, use this form.
- `create_hard_link(`**`tgt, lnk,`** `[`**`ec`**`])` creates a hard link to **`tgt`** at **`lnk`**.
- `create_symlink(`**`tgt, lnk,`** `[`**`ec`**`])` creates a symlink to **`tgt`** at **`lnk`**.
- `create_directory_symlink(`**`tgt, lnk,`** `[`**`ec`**`])` should be used for directories instead of `create_symlink`.
- `remove(`**`p,`** `[`**`ec`**`])` removes a file or empty directory **`p`** (without following symlinks).
- `remove_all(`**`p,`** `[`**`ec`**`])` removes a file or directory recursively **`p`** (without following symlinks).
- `rename(`**`p1, p2,`** `[`**`ec`**`])` renames **`p1`** to **`p2`**.
- `resize_file(`**`p, new_size,`** `[`**`ec`**`])` changes the size of **`p`** (if it's a regular file) to **`new_size`**. If this operation grows the file, zeros fill the new space. Otherwise, the operation trims **`p`** from the end.

You can create a program that copies, resizes, and deletes a file using several of these methods. Listing 17-6 illustrates this by defining a function that prints file size and modification time. In `main`, the program creates and modifies two `path` objects, and it invokes that function after each modification.

```
#include <iostream>
#include <filesystem>

using namespace std;
using namespace std::filesystem;
using namespace std::chrono;
```

```
void write_info(const path& p) {
  if (!exists(p)) { ❶
    cout << p << " does not exist." << endl;
    return;
  }
  const auto last_write = last_write_time(p).time_since_epoch();
  const auto in_hours = duration_cast<hours>(last_write).count();
  cout << p << "\t" << in_hours << "\t" << file_size(p) << "\n"; ❷
}

int main() {
  const path win_path{ R"(C:/Windows/System32/kernel32.dll)" }; ❸
  const auto reamde_path = temp_directory_path() / "REAMDE"; ❹
  try {
    write_info(win_path); ❺
    write_info(reamde_path); ❻

    cout << "Copying " << win_path.filename()
         << " to " << reamde_path.filename() << "\n";
    copy_file(win_path, reamde_path);
    write_info(reamde_path); ❼

    cout << "Resizing " << reamde_path.filename() << "\n";
    resize_file(reamde_path, 1024);
    write_info(reamde_path); ❽

    cout << "Removing " << reamde_path.filename() << "\n";
    remove(reamde_path);
    write_info(reamde_path); ❾
  } catch(const exception& e) {
    cerr << "Exception: " << e.what() << endl;
  }
}
-------------------------------------------------------------------------------
"C:/Windows/System32/kernel32.dll"      3657767 720632 ❺
"C:\\Users\\lospi\\AppData\\Local\\Temp\\REAMDE" does not exist. ❻
Copying "kernel32.dll" to "REAMDE"
"C:\\Users\\lospi\\AppData\\Local\\Temp\\REAMDE"       3657767 720632 ❼
Resizing "REAMDE"
"C:\\Users\\lospi\\AppData\\Local\\Temp\\REAMDE"       3659294 1024 ❽
Removing "REAMDE"
"C:\\Users\\lospi\\AppData\\Local\\Temp\\REAMDE" does not exist. ❾
```

Listing 17-6: A program illustrating several methods for interacting with the filesystem. (Output is from a Windows 10 x64 system.)

The `write_info` function takes a single `path` parameter. You check whether this path exists ❶, printing an error message and returning immediately if it doesn't. If the `path` does exist, you print a message indicating its last modification time (in hours since epoch) and its file size ❷.

Within `main`, you create a path `win_path` to `kernel32.dll` ❸ and a path to a nonexistent file called `REAMDE` in the filesystem's temporary file directory at `reamde_path` ❹. (Recall from Table 17-1 that you can use `operator/` to concatenate two path objects.) Within a try-catch block, you invoke write_info on both paths ❺❻. (If you're using a non-Windows machine, you'll get different output. You can modify win_path to an existing file on your system to follow along.)

Next, you copy the file at win_path to reamde_path and invoke write_info on it ❼. Notice that, as opposed to earlier ❻, the file at reamde_path exists and it has the same last write time and file size as kernel32.dll.

You then resize the file at reamde_path to 1024 bytes and invoke write_info ❽. Notice that the last write time increased from 3657767 to 3659294 and the file size decreased from 720632 to 1024.

Finally, you remove the file at reamde_path and invoke write_info ❾, which tells you that the file again no longer exists.

NOTE *How filesystems resize files behind the scenes varies by operating system and is beyond the scope of this book. But you can think of how a resize operation might work conceptually as the* resize *operation on a* std::vector*. All the data at the end of the file that doesn't fit into the file's new size is discarded by the operating system.*

Directory Iterators

The Filesystem library provides two classes for iterating over the elements of a directory: std::filesystem::directory_iterator and std::filesystem::recursive_directory_iterator. A directory_iterator won't enter subdirectories, but the recursive_directory_iterator will. This section introduces the directory_iterator, but the recursive_directory_iterator is a drop-in replacement and supports all the following operations.

Constructing

The default constructor of directory_iterator produces the end iterator. (Recall that an input end iterator indicates when an input range is exhausted.) Another constructor accepts path, which indicates the directory you want to enumerate. Optionally, you can provide std::filesystem::directory_options, which is an enum class bitmask with the following constants:

- none directs the iterator to skip directory symlinks. If the iterator encounters a permission denial, it produces an error.
- follow_directory_symlink follows symlinks.
- skip_permission_denied skips directories if the iterator encounters a permission denial.

Additionally, you can provide a std::error_code, which, like all other Filesystem library functions that accept an error_code, will set this parameter rather than throwing an exception if an error occurs during construction.

Table 17-2 summarizes these options for constructing a directory_iterator. Note that p is path and d is directory, op is directory_options, and ec is error_code in the table.

Table 17-2: A Summary of `std::filesystem::directory_iterator` Operations

Operation	Notes
`directory_iterator{}`	Constructs the end iterator.
`directory_iterator{ p, [op], [ec] }`	Constructs a directory iterator referring to the directory **p**. The argument **op** defaults to none. If provided, **ec** receives error conditions rather than throwing an exception.
`directory_iterator { d }` `d1 = d2`	Copies construction/assignment.
`directory_iterator { move(d) }` `d1 = move(d2)`	Moves construction/assignment.

Directory Entries

The input iterators `directory_iterator` and `recursive_directory_iterator` produce a `std::filesystem::directory_entry` element for each entry they encounter. The `directory_entry` class stores a `path`, as well as some attributes about that `path` exposed as methods. Table 17-3 lists these methods. Note that `de` is a `directory_entry` in the table.

Table 17-3: A Summary of `std::filesystem::directory_entry` Operations

Operation	Description
`de.path()`	Returns the referenced path.
`de.exists()`	Returns `true` if the referenced path exists on the filesystem.
`de.is_block_file()`	Returns `true` if the referenced path is a block device.
`de.is_character_file()`	Returns `true` if the referenced path is a character device.
`de.is_directory()`	Returns `true` if the referenced path is a directory.
`de.is_fifo()`	Returns `true` if the referenced path is a named pipe.
`de.is_regular_file()`	Returns `true` if the referenced path is a regular file.
`de.is_socket()`	Returns `true` if the referenced path is a socket.
`de.is_symlink()`	Returns `true` if the referenced path is a symlink
`de.is_other()`	Returns `true` if the referenced path is something else.
`de.file_size()`	Returns the size of the referenced path.
`de.hard_link_count()`	Returns the number of hard links to the referenced path.
`de.last_write_time([t])`	If **t** is provided, sets the last modified time of the referenced path; otherwise, it returns the last modified time.
`de.status()` `de.symlink_status()`	Returns a `std::filesystem::file_status` for the referenced path.

You can employ `directory_iterator` and several of the operations in Table 17-3 to create a simple directory-listing program, as Listing 17-7 illustrates.

```
#include <iostream>
#include <filesystem>
#include <iomanip>

using namespace std;
using namespace std::filesystem;
using namespace std::chrono;

void describe(const directory_entry& entry) { ❶
  try {
    if (entry.is_directory()) { ❷
      cout << "           *";
    } else {
      cout << setw(12) << entry.file_size();
    }
    const auto lw_time =
      duration_cast<seconds>(entry.last_write_time().time_since_epoch());
    cout << setw(12) << lw_time.count()
      << " " << entry.path().filename().string()
      << "\n"; ❸
  } catch (const exception& e) {
    cout << "Error accessing " << entry.path().string()
         << ": " << e.what() << endl; ❹
  }
}

int main(int argc, const char** argv) {
  if (argc != 2) {
    cerr << "Usage: listdir PATH";
    return -1; ❺
  }
  const path sys_path{ argv[1] }; ❻
  cout << "Size         Last Write  Name\n";
  cout << "------------ ----------- ------------\n"; ❼
  for (const auto& entry : directory_iterator{ sys_path }) ❽
    describe(entry); ❾
}
```

```
> listdir c:\Windows
Size         Last Write  Name
------------ ----------- ------------
           * 13177963504 addins
           * 13171360979 appcompat
--snip--
           * 13173551028 WinSxS
      316640 13167963236 WMSysPr9.prx
       11264 13167963259 write.exe
```

Listing 17-7: A file- and directory-listing program that uses `std::filesystem::directory_iterator` to enumerate a given directory. (Output is from a Windows 10 x64 system.)

NOTE *You should modify the program's name from `listdir` to whatever value matches your compiler's output.*

You first define a `describe` function that takes a `path` reference ❶, which checks whether the path is a directory ❷ and prints an asterisk for a directory and a corresponding size for a file. Next, you determine the entry's last modification in seconds since epoch and print it along with the entry's associated filename ❸. If any exception occurs, you print an error message and return ❹.

Within `main`, you first check that the user invoked your program with a single argument and return with a negative number if not ❺. Next, you construct a path using the single argument ❻, print some fancy headers for your output ❼, iterate over each `entry` in the directory ❽, and pass it to `describe` ❾.

Recursive Directory Iteration

The `recursive_directory_iterator` is a drop-in replacement for `directory_iterator` in the sense that it supports all the same operations but will enumerate subdirectories. You can use these iterators in combination to build a program that computes the size and quantity of files and subdirectories for a given directory. Listing 17-8 illustrates how.

```
#include <iostream>
#include <filesystem>

using namespace std;
using namespace std::filesystem;

struct Attributes {
  Attributes& operator+=(const Attributes& other) {
    this->size_bytes += other.size_bytes;
    this->n_directories += other.n_directories;
    this->n_files += other.n_files;
    return *this;
  }
  size_t size_bytes;
  size_t n_directories;
  size_t n_files;
}; ❶

void print_line(const Attributes& attributes, string_view path) {
  cout << setw(14) << attributes.size_bytes
       << setw(7) << attributes.n_files
       << setw(7) << attributes.n_directories
       << " " << path << "\n"; ❷
}

Attributes explore(const directory_entry& directory) {
  Attributes attributes{};
  for(const auto& entry : recursive_directory_iterator{ directory.path() }) { ❸
      if (entry.is_directory()) {
        attributes.n_directories++; ❹
      } else {
        attributes.n_files++;
```

```
        attributes.size_bytes += entry.file_size(); ❺
      }
    }
  return attributes;
}

int main(int argc, const char** argv) {
  if (argc != 2) {
    cerr << "Usage: treedir PATH";
    return -1; ❻
  }
  const path sys_path{ argv[1] };
  cout << "Size           Files  Dirs   Name\n";
  cout << "-------------- ------ ------ ------------\n";
  Attributes root_attributes{};
  for (const auto& entry : directory_iterator{ sys_path }) { ❼
    try {
      if (entry.is_directory()) {
        const auto attributes = explore(entry); ❽
        root_attributes += attributes;
        print_line(attributes, entry.path().string());
        root_attributes.n_directories++;
      } else {
        root_attributes.n_files++;
        error_code ec;
        root_attributes.size_bytes += entry.file_size(ec); ❾
        if (ec) cerr << "Error reading file size: "
                     << entry.path().string() << endl;
      }
    } catch(const exception&) {
    }
  }
  print_line(root_attributes, argv[1]); ❿
}
```

```
> treedir C:\Windows
Size         Files  Dirs Name
------------ ----- ----- ------------
         802      1      0 C:\Windows\addins
     8267330      9      5 C:\Windows\apppatch
--snip--
   11396916465  73383  20480 C:\Windows\WinSxS
   21038460348 110950  26513 C:\Windows ❿
```

Listing 17-8: A file- and directory-listing program that uses `std::filesystem::recursive_directory_iterator` to list the number of files and total size of a given path's subdirectory. (Output is from a Windows 10 x64 system.)

NOTE *You should modify the program's name from `treedir` to whatever value matches your compiler's output.*

After declaring the `Attributes` class for storing accounting data ❶, you define a `print_line` function that presents an `Attributes` instance in a user-friendly way alongside a path string ❷. Next, you define an `explore` function that accepts a `directory_entry` reference and iterates over it recursively ❸. If the resulting `entry` is a directory, you increment the directory count ❹; otherwise, you increment the file count and total size ❺.

Within `main`, you check that the program invoked with exactly two arguments. If not, you return with an error code -1 ❻. You employ a (non-recursive) `directory_iterator` to enumerate the contents of the target path referred by `sys_path` ❼. If an `entry` is a directory, you invoke `explore` to determine its attributes ❽, which you subsequently print to the console. You also increment the `n_directories` member of `root_attributes` to keep account. If the `entry` isn't a directory, you add to the `n_files` and `size_bytes` members of `root_attributes` accordingly ❾.

Once you've completed iterating over all `sys_path` subelements, you print `root_attributes` as the final line ❿. The final line of output in Listing 17-8, for example, shows that this particular Windows directory contains 110,950 files occupying 21,038,460,348 bytes (about 21GB) and 26,513 subdirectories.

fstream Interoperation

You can construct file streams (`basic_ifstream`, `basic_ofstream`, or `basic_fstream`) using `std::filesystem::path` or `std::filesystem::directory_entry` in addition to string types.

For example, you can iterate over a directory and construct an `ifstream` to read each file you encounter. Listing 17-9 illustrates how to check for the magic `MZ` bytes at the beginning of each Windows portable executable file (a *.sys*, a *.dll*, a *.exe*, and so on) and report any file that violates this rule.

```
#include <iostream>
#include <fstream>
#include <filesystem>
#include <unordered_set>

using namespace std;
using namespace std::filesystem;

int main(int argc, const char** argv) {
  if (argc != 2) {
    cerr << "Usage: pecheck PATH";
    return -1; ❶
  }
  const unordered_set<string> pe_extensions{
    ".acm", ".ax",  ".cpl", ".dll", ".drv",
    ".efi", ".exe", ".mui", ".ocx", ".scr",
    ".sys", ".tsp"
  }; ❷
  const path sys_path{ argv[1] };
  cout << "Searching " << sys_path << " recursively.\n";
  size_t n_searched{};
  auto iterator = recursive_directory_iterator{ sys_path,
                               directory_options::skip_permission_denied }; ❸
  for (const auto& entry : iterator) { ❹
    try {
```

```
      if (!entry.is_regular_file()) continue;
      const auto& extension = entry.path().extension().string();
      const auto is_pe = pe_extensions.find(extension) != pe_extensions.end();
      if (!is_pe) continue; ❺
      ifstream file{ entry.path() }; ❻
      char first{}, second{};
      if (file) file >> first;
      if (file) file >> second; ❼
      if (first != 'M' || second != 'Z')
        cout << "Invalid PE found: " << entry.path().string() << "\n"; ❽
      ++n_searched;
    } catch(const exception& e) {
      cerr << "Error reading " << entry.path().string()
           << ": " << e.what() << endl;
    }
  }
  cout << "Searched " << n_searched << " PEs for magic bytes." << endl; ❾
}
--------------------------------------------------------------------------------
listing_17_9.exe c:\Windows\System32
Searching "c:\\Windows\\System32" recursively.
Searched 8231 PEs for magic bytes.
```

Listing 17-9: Searching the Windows System32 directory for Windows portable executable files

In `main`, you check for exactly two arguments and return an error code as appropriate ❶. You construct an `unordered_set` containing all the extensions associated with portable executable files ❷, which you'll use to check file extensions. You use a `recursive_directory_iterator` with the `directory_options::skip_permission_denied` option to enumerate all the files in the specified path ❸. You iterate over each entry ❹, skipping over anything that's not a regular file, and you determine whether the entry is a portable executable by attempting to `find` it in `pe_extensions`. If the entry doesn't have such an extension, you skip over the file ❺.

To open the file, you simply pass the path of the `entry` into the constructor of `ifstream` ❻. You then use the resulting input file stream to read the first two bytes of the file into `first` and `second` ❼. If these first two characters aren't `MZ`, you print a message to the console ❽. Either way, you increment a counter called `n_searched`. After exhausting the directory iterator, you print a message indicating `n_searched` to the user before returning from `main` ❾.

Summary

In this chapter, you learned about the stdlib filesystem facilities, including paths, files, directories, and error handling. These facilities enable you to write cross-platform code that interacts with the files in your environment. The chapter culminated with some important operations, directory iterators, and interoperation with file streams.

EXERCISES

17-1. Implement a program that takes two arguments: a path and an extension. The program should search the given path recursively and print any file with the specified extension.

17-2. Improve the program in Listing 17-8 so it can take an optional second argument. If the first argument begins with a hyphen (-), the program reads all contiguous letters immediately following the hyphen and parses each letter as an option. The second argument then becomes the path to search. If the list of options contains an *R*, perform a recursive directory. Otherwise, don't use a recursive directory iterator.

17-3. Refer to the documentation for the *dir* or *ls* command and implement as many of the options as possible in your new, improved version of Listing 17-8.

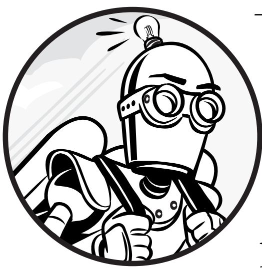
FURTHER READING

- *Windows NT File System Internals: A Developer's Guide* by Rajeev Nagar (O'Reilly, 1997)
- *The Boost C++ Libraries*, 2nd Edition, by Boris Schäling (XML Press, 2014)
- *The Linux Programming Interface: A Linux and UNIX System Programming Handbook* by Michael Kerrisk (No Starch Press, 2010)

18

ALGORITHMS

And that's really the essence of programming. By the time you've sorted out a complicated idea into little steps that even a stupid machine can deal with, you've learned something about it yourself.
—Douglas Adams, Dirk Gently's Holistic Detective Agency



An *algorithm* is a procedure for solving a class of problems. The stdlib and Boost libraries contain a multitude of algorithms that you can use in your programs. Because many very smart people have put a lot of time into ensuring these algorithms are correct and efficient, you should usually not attempt to, for example, write your own sorting algorithm.

Because this chapter covers almost the entire stdlib algorithm suite, it's lengthy; however, the individual algorithm presentations are succinct. On first reading, you should skim through each section to survey the wide range of algorithms available to you. Don't try to memorize them. Instead, focus on getting insight into the kinds of problems you can solve with them as you write code in the future. That way, when you need to use an algorithm, you can say, "Wait, didn't someone already invent this wheel?"

Before you begin working with the algorithms, you'll need some grounding in complexity and parallelism. These two algorithmic characteristics are the main drivers behind how your code will perform.

Algorithmic Complexity

Algorithmic complexity describes the difficulty of a computational task. One way to quantify this complexity is with *Bachmann-Landau* or *"Big O" notation*. Big O notation characterizes functions according to how computation grows with respect to the size of input. This notation only includes the leading term of the complexity function. The *leading term* is the one that grows most quickly as input size increases.

For example, an algorithm whose complexity increases by roughly a fixed amount for each additional input element has a Big O notation of **O(N)**, whereas an algorithm whose complexity doesn't change given additional input has a Big O notation of **O(1)**.

This chapter characterizes the stdlib's algorithms that fall into five complexity classes, as outlined in the list that follows. To give you some idea of how these algorithms scale, each class is listed with its Big O notation and an idea of roughly how many additional operations would be required due to the leading term when input increases from 1,000 elements to 10,000 elements. Each example provides an operation with the given complexity class, where *N* is the number of elements involved in the operation:

Constant time O(1) No additional computation. An example is determining the size of a `std::vector`.

Logarithmic time O(log *N*) About one additional computation. An example is finding an element in a `std::set`.

Linear time O(*N*) About 9,000 additional computations. An example is summing all the elements in a collection.

Quasilinear time O(*N* log *N*) About 37,000 additional computations. An example is quicksort, a commonly used sorting algorithm.

Polynomial (or quadratic) time O(N^2) About 99,000,000 additional computations. An example is comparing all the elements in a collection with all the elements in another collection.

An entire field of computer science is dedicated to classifying computational problems according to their difficulty, so this is an involved topic. This chapter mentions each algorithm's complexity according to how the size of the target sequence affects the amount of required work. In practice, you should profile performance to determine whether an algorithm has suitable scaling properties. But these complexity classes can give you a sense of how expensive a particular algorithm is.

Execution Policies

Some algorithms, those that are commonly called *parallel algorithms*, can divide an algorithm so that independent entities can work on different parts of the problem simultaneously. Many stdlib algorithms allow you to specify parallelism with an *execution policy*. An execution policy indicates the allowed parallelism for an algorithm. From the stdlib's perspective, an algorithm can be executed either *sequentially* or *in parallel*. A sequential algorithm can have only a single entity working on the problem at a time; a parallel algorithm can have many entities working in concert to resolve the problem.

In addition, parallel algorithms can either be *vectorized* or *non-vectorized*. Vectorized algorithms allow entities to perform work in an unspecified order, even allowing a single entity to work on multiple portions of the problem simultaneously. For example, an algorithm that requires synchronization among entities is usually non-vectorizable because the same entity could attempt to acquire a lock multiple times, resulting in a deadlock.

Three execution policies exist in the `<execution>` header:

- `std::execution::seq` specifies sequential (not parallel) execution.
- `std::execution::par` specifies parallel execution.
- `std::execution::par_unseq` specifies parallel *and* vectorized execution.

For those algorithms that support an execution policy, the default is `seq`, meaning you have to opt into parallelism and the associated performance benefits. Note that the C++ Standard doesn't specify the precise meaning of these execution policies because different platforms handle parallelism differently. When you provide a non-sequential execution policy, you're simply declaring that "this algorithm is safe to parallelize."

In Chapter 19, you'll explore execution policies in greater detail. For now, just note that some algorithms permit parallelism.

WARNING *The algorithm descriptions in this chapter aren't complete. They contain enough information to give you a good background on many algorithms available to you in the Standard library. I suggest that, once you've identified an algorithm that fits your needs, you look at one of the resources in the "Further Reading" section at the end of this chapter. Algorithms that accept an optional execution policy often have different requirements when non-default policies are provided, especially where iterators are concerned. For example, if an algorithm normally takes an input iterator, using an execution policy will typically cause the algorithm to require forward iterators instead. Listing these differences would lengthen an already prodigious chapter, so the descriptions omit them.*

> **HOW TO USE THIS CHAPTER**
>
> This chapter is a quick reference that contains more than 50 algorithms. Coverage of each algorithm is necessarily succinct. Each algorithm begins with a terse description. A shorthand representation of the algorithm's function declaration follows along with an explanation of each argument. The declaration depicts optional arguments in brackets. Next, the listing displays the algorithmic complexity. The listing concludes with a non-exhaustive but illustrative example that employs the algorithm. Almost all examples in this chapter are unit tests and implicitly include the following frontmatter:
>
> ```
> #include "catch.hpp"
> #include <vector>
> #include <string>
>
> using namespace std;
> ```
>
> Refer to the relevant subsection [algorithms] for algorithm details should you need them.

Non-Modifying Sequence Operations

A *non-modifying sequence operation* is an algorithm that performs computation over a sequence but doesn't modify the sequence in any way. You can think of these as `const` algorithms. Each algorithm explained in this section is in the `<algorithm>` header.

all_of

The `all_of` algorithm determines whether each element in a sequence meets some user-specified criteria.

The algorithm returns `true` if the target sequence is empty or if `pred` is `true` for *all* elements in the sequence; otherwise, it returns `false`.

```
bool all_of([ep], ipt_begin, ipt_end, pred);
```

Arguments

- An optional `std::execution` execution policy, `ep` (default: `std::execution::seq`)
- A pair of `InputIterator` objects, `ipt_begin` and `ipt_end`, representing the target sequence
- A unary predicate, `pred`, that accepts an element from the target sequence

Complexity

Linear The algorithm invokes pred at most distance(ipt_begin, ipt_end) times.

Examples

```
#include <algorithm>

TEST_CASE("all_of") {
  vector<string> words{ "Auntie", "Anne's", "alligator" }; ❶
  const auto starts_with_a =
    [](const auto& word❷) {
      if (word.empty()) return false; ❸
      return word[0] == 'A' || word[0] == 'a'; ❹
    };
  REQUIRE(all_of(words.cbegin(), words.cend(), starts_with_a)); ❺
  const auto has_length_six = [](const auto& word) {
    return word.length() == 6; ❻
  };
  REQUIRE_FALSE(all_of(words.cbegin(), words.cend(), has_length_six)); ❼
}
```

After constructing a vector containing string objects called words ❶, you construct the lambda predicate starts_with_a, which takes a single object called word ❷. If word is empty, starts_with_a returns false ❸; otherwise, it returns true if word starts with either a or A ❹. Because all of the word elements start with either a or A, all_of returns true when it applies starts_with_a ❺.

In the second example, you construct the predicate has_length_six, which returns true only if word has length six ❻. Because alligator doesn't have length six, all_of returns false when it applies has_length_six to words ❼.

any_of

The any_of algorithm determines whether any element in a sequence meets some user-specified criteria.

The algorithm returns false if the target sequence is empty or if pred is true for *any* element in the sequence; otherwise, it returns false.

```
bool any_of([ep], ipt_begin, ipt_end, pred);
```

Arguments

- An optional std::execution execution policy, ep (default: std::execution ::seq)
- A pair of InputIterator objects, ipt_begin and ipt_end, representing the target sequence
- A unary predicate, pred, that accepts an element from the target sequence

Complexity

Linear The algorithm invokes pred at most distance(ipt_begin, ipt_end) times.

Examples

```
#include <algorithm>

TEST_CASE("any_of") {
  vector<string> words{ "Barber", "baby", "bubbles" }; ❶
  const auto contains_bar = [](const auto& word) {
    return word.find("Bar") != string::npos;
  }; ❷
  REQUIRE(any_of(words.cbegin(), words.cend(), contains_bar)); ❸

  const auto is_empty = [](const auto& word) { return word.empty(); }; ❹
  REQUIRE_FALSE(any_of(words.cbegin(), words.cend(), is_empty)); ❺
}
```

After constructing a vector containing string objects called words ❶, you construct the lambda predicate contains_bar that takes a single object called word ❷. If word contains the substring Bar, it returns true; otherwise, it returns false. Because Barber contains Bar, any_of returns true when it applies contains_bar ❸.

In the second example, you construct the predicate is_empty, which returns true only if a word is empty ❹. Because none of the words are empty, any_of returns false when it applies is_empty to words ❺.

none_of

The none_of algorithm determines whether no element in a sequence meets some user-specified criteria.

The algorithm returns true if the target sequence is empty or if pred is true for *no* element in the sequence; otherwise, it returns false.

```
bool none_of([ep], ipt_begin, ipt_end, pred);
```

Arguments

- An optional std::execution execution policy, ep (default: std::execution ::seq)
- A pair of InputIterator objects, ipt_begin and ipt_end, representing the target sequence
- A unary predicate, pred, that accepts an element from the target sequence

Complexity

Linear The algorithm invokes pred at most distance(ipt_begin, ipt_end) times.

Examples

```
#include <algorithm>

TEST_CASE("none_of") {
  vector<string> words{ "Camel", "on", "the", "ceiling" }; ❶
  const auto is_hump_day = [](const auto& word) {
    return word == "hump day";
  }; ❷
  REQUIRE(none_of(words.cbegin(), words.cend(), is_hump_day)); ❸

  const auto is_definite_article = [](const auto& word) {
    return word == "the" || word == "ye";
  }; ❹
  REQUIRE_FALSE(none_of(words.cbegin(), words.cend(), is_definite_article)); ❺
}
```

After constructing a `vector` containing `string` objects called `words` ❶, you construct the lambda predicate `is_hump_day` that takes a single object called `word` ❷. If `word` equals `hump day`, it returns `true`; otherwise, it returns `false`. Because `words` doesn't contain `hump day`, `none_of` returns `true` when it applies `is_hump_day` ❸.

In the second example, you construct the predicate `is_definite_article`, which returns `true` only if `word` is a definite article ❹. Because `the` is a definite article, `none_of` returns `false` when it applies `is_definite_article` to `words` ❺.

for_each

The `for_each` algorithm applies some user-defined function to each element in a sequence.

The algorithm applies `fn` to each element of the target sequence. Although `for_each` is considered a non-modifying sequence operation, if `ipt_begin` is a mutable iterator, `fn` can accept a non-`const` argument. Any values that `fn` returns are ignored.

If you omit `ep`, `for_each` will return `fn`. Otherwise, `for_each` returns `void`.

```
for_each([ep], ipt_begin, ipt_end, fn);
```

Arguments

- An optional `std::execution` execution policy, `ep` (default: `std::execution::seq`)
- A pair of `InputIterator` objects, `ipt_begin` and `ipt_end`, representing the target sequence
- A unary function, `fn`, that accepts an element from the target sequence