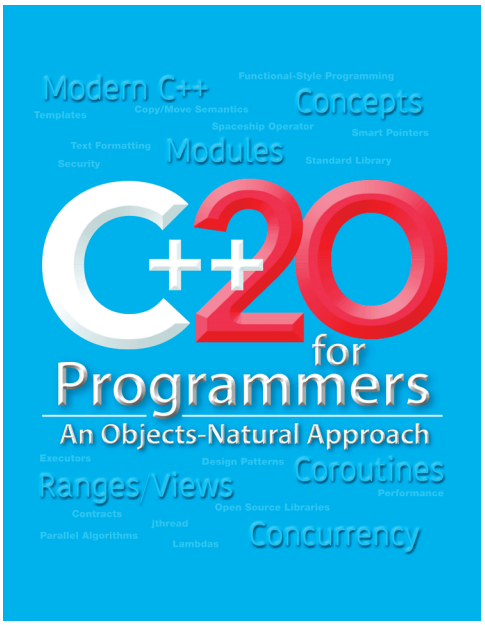
19

Stream I/O & C++20 Text Formatting

Objectives

In this chapter, you'll:

- Use C++ object-oriented stream input/output.
- Input and output individual characters.
- Use unformatted I/O for high performance.
- Use stream manipulators to display integers in octal and hexadecimal formats.
- Specify precision for input and output.
- Display floating-point values in scientific and fixed-point notation.
- Set and restore the format state.
- Control alignment and padding.
- Determine the success or failure of input/output operations.
- Tie output streams to input streams.
- See many of C++20's concise and convenient text-formatting capabilities, including presentation types to specify data types to format, positional arguments, field widths, alignment, numeric formatting and using placeholders to specify field widths and precisions.



Outline

19.1 Introduction

This chapter discusses input/output formatting capabilities. First, we present the I/O streams formatting capabilities that you're likely to see in C++ legacy code. Then, we discuss features from C++20's new text-formatting capabilities. In Section 19.9, you'll see that C++20 text formatting is more concise and convenient than the I/O streams formatting capabilities presented in Section 19.6.

C++ uses **type-safe I/O**. Each I/O operation is executed in a manner sensitive to the data type. If an I/O function has been defined to handle a particular data type, then that function is called to handle that data type. If there is no match between the type of the actual data and a function for handling data of that type, the compiler generates an error. Thus, improper data cannot "sneak" through the system. As you saw in Chapter 11, you can specify how to perform I/O for objects of user-defined types by overloading the stream insertion operator (<<) and the stream extraction operator (>>).

19.2 Streams

C++ I/O occurs in **streams**, which are sequences of bytes. In input operations, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection) to main memory. In output operations, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection).

An application associates meaning with bytes. The bytes could represent characters, raw data, graphics images, audio, video or other information an application requires. The system I/O mechanisms transfer bytes from devices to memory and vice versa. The time these transfers take typically is far greater than the time the processor requires to manipulate data in memory. I/O operations require careful planning and tuning to ensure optimal performance. Perf

C++ provides both "low-level" and "high-level" I/O capabilities. Low-level **unformatted I/O** capabilities specify that some number of bytes should be transferred device-to-memory or memory-to-device. In such transfers, the individual byte is the item of interest. Such low-level capabilities provide high-speed, high-volume transfers but are not particularly convenient. Higher-level **formatted I/O** groups bytes into meaningful units, such as integers, floating-point numbers, characters, strings and custom types.

19.2.1 Classic Streams vs. Standard Streams

C++'s **classic stream libraries** originally supported only `char`-based I/O. Because a `char` occupies one byte, it can represent only a limited set of characters, such as those in the ASCII character set. Many languages use alphabets that contain more characters than a single-byte `char` can represent. Such characters are typically available in the extensive international **Unicode® character set** (`https://unicode.org`), which can represent most of the world's languages, mathematical symbols, emoji characters and more. C++ supports Unicode via the types

- `wchar_t`,
- `char16_t` and `char32_t` (both from C++11), and 11
- `char8_t` (C++20). 20

The **standard stream library classes** are class templates that can be instantiated for these various character types. We use the predefined stream-library instantiations for type `char` in this book. Unicode-based applications would use appropriate class-template instantiations based on the preceding types. C++ also supports Unicode string literals—for more information, see

```
https://en.cppreference.com/w/cpp/language/string_literal
```

19.2.2 `iostream` Library Headers

The C++ stream libraries provide hundreds of I/O capabilities. Most of our C++ programs include the `<iostream>` header, which declares basic services required for all stream-I/O operations. The `<iostream>` header defines the **`cin`**, **`cout`**, **`cerr`** and **`clog`** objects, which correspond to the **standard input stream**, the **standard output stream**, the **unbuffered standard error stream** and the **buffered standard error stream**, respectively. The next section discusses `cerr`, `clog` and buffering. The `<iostream>` and `<iomanip>` headers define **stream manipulators** for formatted I/O. We'll demonstrate many of these in this chapter. We'll also show that the newer C++20 text formatting with the `format` function greatly 20 simplifies formatting output.

19.2.3 Stream Input/Output Classes and Objects

This chapter focuses on the `iostream` class templates:

- `basic_istream` for stream input operations and
- `basic_ostream` for stream output operations.

Though we do not use it in this chapter, `basic_iostream` provides stream input and stream output operations.

For each of the class templates `basic_istream`, `basic_ostream` and `basic_iostream`, the `iostream` library defines a type alias for `char`-based I/O:

- `istream` is a `basic_istream<char>` for `char` input—this is `cin`'s type.
- `ostream` is a `basic_ostream<char>` for `char` output—this is the type of `cout`, `cerr` and `clog`.
- `iostream` is a `basic_iostream<char>` for `char` input and output.

We used the aliases `istream` and `ostream` in Chapter 11 when we overloaded the stream extraction and stream insertion operators. The library also defines versions of these for `wchar_t`-based I/O—named `wistream`, `wostream` and `wiostream`, respectively. We cover only the `char`-based streams here.

Standard Stream Objects `cin`, `cout`, `cerr` and `clog`

Predefined object `cin` is an `istream` that's connected to the **standard input device**—usually the keyboard. In a stream extraction operation (>>) like

```
int grade{0};
std::cin >> grade; // data "flows" in the direction of the arrows
```

the compiler selects the appropriate overloaded stream extraction operator, based on the type of the variable `grade`—the one for `int` in this case. The >> operator is overloaded to input fundamental-type values, strings and pointer values.

The predefined object `cout` is an `ostream` that's connected to the **standard output device**. Standard output typically appears in

- a Command Prompt or PowerShell window in Microsoft Windows,
- a Terminal in macOS or Linux, or
- a shell window in Linux.

The stream insertion operator (<<) outputs its right operand to the standard output device:

```
std::cout << grade; // data "flows" in the direction of the arrows
```

The compiler selects the appropriate stream insertion operator for `grade`'s type—<< is overloaded to output data items of fundamental types, strings and pointer values.

The predefined object `cerr` is an `ostream` that's connected to the **standard error device**—typically the same device as the standard output device. Outputs to object `cerr` are **unbuffered**, meaning that each stream insertion to `cerr` performs its output immediately—this is appropriate for notifying a user promptly about errors.

The predefined object `clog` is an `ostream` that's connected to the **standard error device**. Outputs to `clog` are **buffered**. Each output might be held in a buffer (that is, an area in memory) until the buffer is filled or until the buffer is flushed. Buffering is an I/O performance-enhancement technique.

Perf

19.3 Stream Output

`ostream` provides both formatted and unformatted output capabilities, including

- outputting standard data types with the stream insertion operator (<<);
- outputting characters via the `put` member function;
- **unformatted output** via the `write` member function;
- outputting integers in decimal, octal and hexadecimal formats;
- outputting floating-point values with various precisions, with **forced decimal points**, in **scientific notation** (e.g., `1.234567e-03`) and in **fixed notation** (e.g., `0.00123457`);
- outputting data **aligned** in fields of designated widths;
- outputting data in fields **padded** with specified characters; and
- outputting uppercase letters in scientific notation and **hexadecimal** (**base-16**) **notation**.

We'll demonstrate all of these capabilities in this chapter.

19.3.1 Output of char* Variables

Generally, you should avoid using pointers in favor of the modern C++ techniques we've discussed. In programs that require pointers, occasionally, you might want to print the addresses they contain (e.g., for debugging). The << operator outputs a `char*` as a **null-terminated C-style string**. To output the **address**, cast the `char*` to a `void*` (Fig. 19.1, line 12). Operator <<'s `void*` version displays the pointer in an implementation-dependent manner—often as a hexadecimal number.[1] Figure 19.1 prints a `char*` variable as a null-terminated C-style string and an address. The output will vary by compiler and operating system. We say more about controlling the bases of numbers in Section 19.6.1 and Section 19.6.8.

```
1  // fig19_01.cpp
2  // Printing the address stored in a char* variable.
3  #include <iostream>
4
5  int main() {
6     const char* const word{"again"};
7
8     // display the value of char* variable word, then display
9     // the value of word after a static_cast to void*
10    std::cout << "Value of word is: " << word
11       << "\nValue of static_cast<const void*>(word) is: "
12       << static_cast<const void*>(word) << '\n';
13 }
```

```
Value of word is: again
Value of static_cast<const void*>(word) is: 00007FF611416410
```

Fig. 19.1 | Printing the address stored in a `char*` variable.

1. To learn more about hexadecimal numbers, see online Appendix C, Number Systems.

19.3.2 Character Output Using Member Function put

The `basic_ostream` member function `put` outputs one character at a time. For example, the statement

```
std::cout.put('A');
```

displays a single character `A`. Calls to `put` can be chained, as in

```
std::cout.put('A').put('\n');
```

which outputs the letter `A` followed by a newline character. As with `<<`, the preceding statement executes in this manner because the dot operator (`.`) groups left-to-right, and the `put` member function returns a reference to the `ostream` object (`cout`) that received the `put` call. You also can call `put` with a numeric expression representing a character value. For example, the following statement outputs uppercase `A`:

```
std::cout.put(65);
```

19.4 Stream Input

`istream` provides formatted and unformatted input capabilities. The stream extraction operator (`>>`) normally skips white-space characters (such as blanks, tabs and newlines) in the input stream. Later, we'll see how to change this behavior.

Using the Result of a Stream Extraction as a Condition

After each input, the stream extraction operator returns a reference to the stream object that received the extraction message (e.g., `cin` in the expression `cin >> grade`). If that reference is used as a condition (e.g., in a `while` statement's loop-continuation condition),
11 the stream's overloaded `bool` cast operator function (C++11) is implicitly invoked to convert the reference into `true` or `false` value, based on the success or failure, respectively, of the last input operation. When an attempt is made to read past the end of a stream, the stream's overloaded `bool` cast operator returns `false` to indicate end-of-file. We used this capability in line 24 of Fig. 4.6.

19.4.1 get and getline Member Functions

The `get` member function with no arguments inputs and returns one character from the designated stream—including white-space characters and other nongraphic characters, such as the key sequence that represents end-of-file. This version of `get` returns `EOF` when end-of-file is encountered on the stream. `EOF` normally has the value `-1` and is defined in a header that's included in your code via stream library headers like `<iostream>`.

Using Member Functions eof, get and put

Figure 19.2 demonstrates member functions `eof` and `get` on input stream `cin` and member function `put` on output stream `cout`. This program uses `get` to read characters into the `int` variable `character`, so we can test for `EOF`. Function `get` returns an `int` because **`char` can represent only nonnegative values on many platforms**, and **`EOF` is typically defined as `-1`**. Line 10 prints the value of `cin.eof()`—initially `false`—before any inputs to show that end-of-file has not yet occurred on `cin`. You enter a line of text and press *Enter* followed by the end-of-file indicator:

- <*Ctrl*> *z* on Microsoft Windows systems or
- <*Ctrl*> *d* on Linux and Mac systems.

Line 14 reads each character, which line 15 outputs to `cout` using member function `put`. When end-of-file is encountered, the `while` statement ends, and lines 19–20 display the integer value of the last character read (`-1` for end-of-file) and the current value of `cin.eof()`, which now returns `true`, to show that end-of-file has been set on `cin`. Function `eof` returns `true` only after the program attempts to read past the last character in the stream.

```
1  // fig19_02.cpp
2  // get, put and eof member functions.
3  #include <format>
4  #include <iostream>
5
6  int main() {
7     int character{0}; // use int, because char cannot represent EOF
8
9     // prompt user to enter line of text
10    std::cout << std::format("Before input, cin.eof(): {}", std::cin.eof())
11       << "\nEnter a sentence followed by Enter and end-of-file:\n";
12
13    // use get to read each character; use put to display it
14    while ((character = std::cin.get()) != EOF) {
15       std::cout.put(character);
16    }
17
18    // display end-of-file character
19    std::cout << std::format("\nEOF on this system is: {}\n", character)
20       << std::format("After EOF input, cin.eof(): {}\n", std::cin.eof());
21 }
```

```
Before input, cin.eof(): false
Enter a sentence followed by Enter and end-of-file:
Testing the get and put member functions
Testing the get and put member functions
^Z

EOF on this system is: -1
After EOF input, cin.eof(): true
```

Fig. 19.2 | `get`, `put` and `eof` member functions.

Other `get` Versions

The `get` member function with a character-reference argument inputs the next character from the input stream and stores it in the character argument. This version of `get` returns a reference to the `istream` object on which the function is invoked.

A third version of `get` takes three arguments—a built-in array of `char`s, a size limit and a delimiter (with default value `'\n'`). This version can read multiple characters from the input stream. It either reads one fewer than the specified maximum number of characters and terminates or terminates as soon as the delimiter is read. A null character is inserted to terminate the input string in the character array argument. The delimiter is not

placed in the character array. Rather, it remains in the input stream and will be the next character read if the program performs more input. Thus, the result of a second consecutive `get` is an empty line (possibly a logic error) unless the delimiter character is removed from the input stream—which you can do simply by calling `cin.ignore()`.

Err

Comparing `cin` and `cin.get`

Figure 19.3 compares input using the stream extraction operator with `cin` (line 14), which reads characters until a white-space character is encountered, and input using the three-argument version of `cin.get` with its third argument defaulted to the `'\n'` character.

```
1  // fig19_03.cpp
2  // Contrasting input of a string via cin and cin.get.
3  #include <format>
4  #include <iostream>
5
6  int main() {
7     // create two char arrays, each with 80 elements
8     constexpr int size{80};
9     char buffer1[size]{};
10    char buffer2[size]{};
11
12    // use cin to input characters into buffer1
13    std::cout << "Enter a sentence:\n";
14    std::cin >> buffer1;
15
16    // display buffer1 contents
17    std::cout << std::format("\nThe cin input was:\n{}\n\n", buffer1);
18
19    // use cin.get to input characters into buffer2
20    std::cin.get(buffer2, size);
21
22    // display buffer2 contents
23    std::cout << std::format("The cin.get input was:\n{}\n", buffer2);
24 }
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The cin input was:
Contrasting

The cin.get input was:
 string input with cin and cin.get
```

Fig. 19.3 | Contrasting input of a string via `cin` and `cin.get`.

Err 20

Before C++20, line 14 could write past the end of `buffer1`—a potentially fatal logic error. In C++20, the `char` array overload of `operator>>` is a **function template** that the compiler instantiates using its `char` array argument's size. In line 14, the compiler knows that `buffer1` contains 80 characters (as defined in line 9), so it instantiates an `operator>>` function that limits the number of characters input to a maximum of 79, saving one array element for the C-style string's terminating `'\0'` character.

Using Member Function `getline`

Member function **`getline`** operates similarly to the third version of the `get` member function and inserts a null character after the line in the built-in array of `char`s. The `getline` function removes the delimiter from the stream (i.e., reads the character and discards it) but **does not store it in the character array**. The program of Fig. 19.4 uses `getline` to input a line of text (line 12). Again, you should avoid using built-in arrays and pointers. So, C++ also provides `std::getline` in the `<string>` header.[2] This version of `getline` reads data and places it into a `string` object.

```
1  // fig19_04.cpp
2  // Inputting characters using cin member function getline.
3  #include <format>
4  #include <iostream>
5
6  int main() {
7     const int size{80};
8     char buffer[size]{}; // create array of 80 characters
9
10    // input characters in buffer via cin function getline
11    std::cout << "Enter a sentence:\n";
12    std::cin.getline(buffer, size);
13
14    // display buffer contents
15    std::cout << std::format("\nYou entered:\n{}\n", buffer);
16 }
```

```
Enter a sentence:
Using the getline member function

You entered:
Using the getline member function
```

Fig. 19.4 | Inputting characters using `cin` member function `getline`.

19.4.2 `istream` Member Functions `peek`, `putback` and `ignore`

`istream` member function `ignore` reads and discards characters. It receives as arguments:

- a designated number of characters—the default argument value is 1—and
- a delimiter at which to stop ignoring characters—the default delimiter is `EOF`.

The function discards the specified number of characters, or fewer characters if the delimiter is encountered in the input stream.

The **`putback`** member function places the previous character obtained by a `get` from an input stream back into that stream. This is helpful in applications that scan an input stream looking for a field beginning with a specific character. When that character is input, the application returns the character to the stream for the next input operation.

The **`peek`** member function returns the next character from an input stream but does not remove the character from the stream.

2. "`std::getline`." Accessed March 29, 2022. `https://en.cppreference.com/w/cpp/string/basic_string/getline`.

19.5 Unformatted I/O Using read, write and gcount

Unformatted input/output is performed using `istream`'s **`read`** and `ostream`'s **`write`** member functions, respectively:

- `read` inputs bytes to a built-in array of `char`s in memory.
- `write` outputs bytes from a built-in array of `char`s.

These bytes are input or output simply as "raw" bytes—they are not formatted in any way. For example, the following call outputs the first 10 bytes of `buffer`, including null characters, if any, that would cause output with `cout` and `<<` to terminate:

```
char buffer[]{"HAPPY BIRTHDAY"};
std::cout.write(buffer, 10);
```

Similarly, the following call displays the first 10 characters of the alphabet:

```
std::cout.write("ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10);
```

The `read` member function inputs a designated number of characters into a built-in array of `char`s. If fewer than the designated number of characters are read, `failbit` is set. Section 19.7 shows how to determine whether `failbit` has been set. Member function **`gcount`** reports the number of characters read by the last input operation.

Figure 19.5 demonstrates `istream` member functions `read` and `gcount`, and `ostream` member function `write`. The program inputs 20 characters (from a longer input sequence) into the array `buffer` with `read` (line 10), determines the number of characters input with `gcount` (line 14) and outputs the characters in `buffer` with `write` (line 14).

```
1  // fig19_05.cpp
2  // Unformatted I/O using read, gcount and write.
3  #include <iostream>
4
5  int main() {
6     char buffer[80]{}; // create array of 80 characters
7
8     // use function read to input characters into buffer
9     std::cout << "Enter a sentence:\n";
10    std::cin.read(buffer, 20);
11
12    // use functions write and gcount to display buffer characters
13    std::cout << "\nThe sentence entered was:\n";
14    std::cout.write(buffer, std::cin.gcount());
15    std::cout << '\n';
16 }
```

```
Enter a sentence:
Using the read, write, and gcount member functions

The sentence entered was:
Using the read, writ
```

Fig. 19.5 | Unformatted I/O using `read`, `gcount` and `write`.

19.6 Stream Manipulators

C++ provides various **stream manipulators** to specify formatting in streams. The stream manipulators provide capabilities such as

- setting the base for integer values
- setting field widths
- setting precision
- setting and unsetting format state
- setting the fill character in fields
- flushing streams
- inserting a newline into the output stream (and flushing the stream)
- inserting a null character into the output stream
- skipping white space in the input stream

The following table lists various stream manipulators that control a given stream's format state. We show examples of many of these stream manipulators in the next several sections.

Manipulator	Description
`skipws`	**Skips white-space characters** on an input stream. You can reset this setting with stream manipulator `noskipws`.
`left`	**Left aligns** output in a field. Padding characters appear to the right if necessary.
`right`	**Right aligns** output in a field. Padding characters appear to the left if necessary.
`internal`	In a field, this **left aligns a number's sign** and **right aligns a number's value**. Padding characters appear between the sign and the number if necessary.
`boolalpha`	Displays `bool` values as the word `true` or `false`. Similarly, `noboolalpha` sets the stream back to displaying bool values as 1 (true) and 0 (false).
`dec`	Treats integers as **decimal** (base 10) values.
`oct`	Treats integers as **octal** (base 8) values.
`hex`	Treats integers as **hexadecimal** (base 16) values.
`showbase`	Outputs a number's **base** before the number—`0` for octals and `0x` or `0X` for hexadecimals. You can reset this with stream manipulator `noshowbase`.
`showpoint`	Forces floating-point numbers do display a decimal point. Normally, this is used with `fixed` to guarantee a certain number of digits to the right of the decimal point. You can reset this setting with stream manipulator `noshowpoint`.
`uppercase`	Displays hexadecimal integers with uppercase letters (i.e., `X` and `A` through `F`) and displays floating-point values in scientific notation with an uppercase `E`. You can reset this setting with stream manipulator `nouppercase`.
`showpos`	Precedes positive numbers by a plus sign (`+`). You can reset this with `noshowpos`.
`scientific`	Outputs floating-point values in **scientific notation**.
`fixed`	Outputs floating-point values in **fixed-point notation** with a specific number of digits to the right of the decimal point.

19.6.1 Integral Stream Base: dec, oct, hex and setbase

Integers typically are processed as decimal (base 10) values. To change this, you can insert the **hex** stream manipulator to set the base to hexadecimal (base 16) or insert the **oct** manipulator to set the base to octal (base 8). Insert the **dec** manipulator to reset the stream base to decimal. These **stream manipulators** are all **sticky**—that is, the settings remain in effect until you change them.

You also can set a stream's integer base via the **setbase parameterized stream manipulator** (header `<iomanip>`). A parameterized stream manipulator takes an argument—in this case, the value `10`, `8`, or `16` to set the base to decimal, octal or hexadecimal.[3] The stream base value remains the same until changed explicitly, so `setbase` settings are sticky. Figure 19.6 demonstrates stream manipulators `hex` (line 14), `dec` (line 17), `oct` (line 18) and `setbase` (line 21).

```
1  // fig19_06.cpp
2  // Using stream manipulators dec, oct, hex and setbase.
3  #include <iomanip>
4  #include <iostream>
5
6  int main() {
7     int number{0};
8
9     std::cout << "Enter a decimal number: ";
10    std::cin >> number; // input number
11
12    // use hex stream manipulator to show hexadecimal number
13    std::cout << number << " in hexadecimal is: "
14       << std::hex << number << "\n";
15
16    // use oct stream manipulator to show octal number
17    std::cout << std::dec << number << " in octal is: "
18       << std::oct << number << "\n";
19
20    // use setbase stream manipulator to show decimal number
21    std::cout << std::setbase(10) << number << " in decimal is: "
22       << number << "\n";
23 }
```

```
Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20
```

Fig. 19.6 | Using stream manipulators `dec`, `oct`, `hex` and `setbase`.

19.6.2 Floating-Point Precision (setprecision, precision)

You can control the **precision** of floating-point numbers—that is, the number of digits to the right of the decimal point—with the **setprecision** stream manipulator or the `ostream` member function **precision**. Both are sticky—a call to either sets the precision for

3. Appendix C, Number Systems, discusses the decimal, octal and hexadecimal number systems.

all subsequent output operations until the next precision-setting call. Calling member function `precision` with no argument returns the current precision setting. You can use this to save the current precision setting so you can restore it later. Figure 19.7 uses both member function `precision` (line 16) and the `setprecision` manipulator (line 24) to print a table that shows the square root of `2`, with precision varying from `0` to `9`. Stream manipulator **`fixed`** (line 12) forces a floating-point number to display in fixed-point notation with a specific number of digits to the right of the decimal point, as specified by member function `precision` or stream manipulator `setprecision`.

```
1  // fig19_07.cpp
2  // Controlling precision of floating-point values.
3  #include <iomanip>
4  #include <iostream>
5  #include <cmath>
6
7  int main() {
8     double root2{std::sqrt(2.0)}; // calculate square root of 2
9
10    std::cout << "Square root of 2 with precisions 0-9.\n"
11       << "Precision set by ostream member function precision:\n";
12    std::cout << std::fixed; // use fixed-point notation
13
14    // display square root using ostream function precision
15    for (int places{0}; places <= 9; ++places) {
16       std::cout.precision(places);
17       std::cout << root2 << "\n";
18    }
19
20    std::cout << "\nPrecision set by stream manipulator setprecision:\n";
21
22    // set precision for each digit, then display square root
23    for (int places{0}; places <= 9; ++places) {
24       std::cout << std::setprecision(places) << root2 << "\n";
25    }
26 }
```

```
Square root of 2 with precisions 0-9.
Precision set by ostream member function precision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 19.7 | Controlling precision of floating-point values. (Part 1 of 2.)

```
Precision set by stream manipulator setprecision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 19.7 | Controlling precision of floating-point values. (Part 2 of 2.)

19.6.3 Field Width (`width`, `setw`)

The **`width`** member function (of classes `istream` and `ostream`) sets the **field width**—that is, the number of character positions in which a value should be output or the maximum number of characters that should be input. The function also returns the previous field width so you can save it and restore the value later. If the values output are narrower than the field width, **fill characters** are inserted as **padding.** When a field is not sufficiently wide to handle outputs, the outputs print as wide as necessary, which can yield confusing outputs. **The width setting is not sticky—it applies only for the next insertion or extraction**. Afterward, the width is set implicitly to `0`, so subsequent inputs or outputs will be performed with default settings. Calling `width` with no argument returns the current setting.

Figure 19.8 demonstrates the `width` member function for both input (lines 10 and 16) and output (line 14). For input into a `char` array, a maximum of **one fewer characters than the width are read**, saving one element for the null character to be placed at the end of the C-style string. Remember that stream extraction terminates when nonleading white space is encountered. When prompted for input in Fig. 19.8, enter a line of text and press *Enter* followed by end-of-file (*<Ctrl> z* on Microsoft Windows systems and *<Ctrl> d* on Linux and OS X systems). The **`setw` parameterized stream manipulator** also may be used to set the field width by inserting a call to it in a `cin` or `cout` statement.

```
1  // fig19_08.cpp
2  // width member function of classes istream and ostream.
3  #include <iostream>
4
5  int main() {
6     int widthValue{4};
7     char sentence[10]{};
8
9     std::cout << "Enter a sentence:\n";
10    std::cin.width(5); // input up to 4 characters from sentence
11
```

Fig. 19.8 | `width` member function of class classes `istream` and `ostream`. (Part 1 of 2.)

```
12      // set field width, then display characters based on that width
13      while (std::cin >> sentence) {
14         std::cout.width(widthValue++);
15         std::cout << sentence << "\n";
16         std::cin.width(5); // input up to 4 more characters from sentence
17      }
18   }
```

```
Enter a sentence:
This is a test of the width member function
This
   is
     a
   test
      of
      the
      widt
          h
        memb
           er
          func
           tion
^Z
```

Fig. 19.8 | `width` member function of class classes `istream` and `ostream`. (Part 2 of 2.)

19.6.4 User-Defined Output Stream Manipulators

You can create your own stream manipulators. Figure 19.9 shows how to create and use custom nonparameterized output-stream manipulators `bell` (lines 7–9) and `tab` (lines 12–14). These are defined as functions with `ostream&` as the return type and parameter type. When lines 19 and 21 insert `tab` and `bell` in the output stream, their corresponding functions are called, which in turn output the `\a` (alert) and `\t` (tab) escape sequences, respectively. The `bell` manipulator does not display any text. Rather, it plays your system's alert sound.

```
1   // fig19_09.cpp
2   // Creating and testing user-defined, nonparameterized
3   // stream manipulators.
4   #include <iostream>
5
6   // bell manipulator (using escape sequence \a)
7   std::ostream& bell(std::ostream& output) {
8      return output << '\a'; // issue system beep
9   }
10
11  // tab manipulator (using escape sequence \t)
12  std::ostream& tab(std::ostream& output) {
13     return output << '\t'; // issue tab
14  }
15
```

Fig. 19.9 | Creating and testing user-defined, nonparameterized stream manipulators. (Part 1 of 2.)

```
16  int main() {
17     // use tab and bell manipulators
18     std::cout << "Testing the tab manipulator:\n"
19        << 'a' << tab << 'b' << tab << 'c' << '\n';
20
21     std::cout << "Testing the bell manipulator\n" << bell;
22  }
```

```
Testing the tab manipulator:
a       b       c
Testing the bell manipulator
```

Fig. 19.9 | Creating and testing user-defined, nonparameterized stream manipulators. (Part 2 of 2.)

19.6.5 Trailing Zeros and Decimal Points (`showpoint`)

Stream manipulator **`showpoint`** (Fig. 19.10) is a sticky setting that forces a floating-point number to display a decimal point and trailing zeros. For example, the floating-point value `79.0` prints as `79` without using `showpoint` and prints as `79.00000` using `showpoint`. The number of trailing zeros is determined by the current **precision**. To reset the `showpoint` setting, output the stream manipulator **`noshowpoint`**. The **default precision** of floating-point numbers is 6, which you can see in the output produced by lines 13–17. When neither the `fixed` nor the `scientific` stream manipulator is used, the precision represents the number of significant digits to display (i.e., the total number of digits to display), not the number of digits to display after the decimal point.

```
1  // fig19_10.cpp
2  // Displaying trailing zeros and decimal points in floating-point values.
3  #include <iostream>
4
5  int main() {
6     // display double values with default stream format
7     std::cout << "Before using showpoint"
8        << "\n9.9900 prints as: " << 9.9900
9        << "\n9.9000 prints as: " << 9.9000
10       << "\n9.0000 prints as: " << 9.0000;
11
12    // display double value after showpoint
13    std::cout << std::showpoint
14       << "\n\nAfter using showpoint"
15       << "\n9.9900 prints as: " << 9.9900
16       << "\n9.9000 prints as: " << 9.9000
17       << "\n9.0000 prints as: " << 9.0000 << '\n';
18 }
```

```
Before using showpoint
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9
```

Fig. 19.10 | Controlling the printing of trailing zeros and decimal points in floating-point values. (Part 1 of 2.)

```
After using showpoint
9.9900 prints as: 9.99000
9.9000 prints as: 9.90000
9.0000 prints as: 9.00000
```

Fig. 19.10 | Controlling the printing of trailing zeros and decimal points in floating-point values. (Part 2 of 2.)

19.6.6 Alignment (`left`, `right` and `internal`)

Stream manipulators `left` and `right` enable fields to be **left-aligned** with **padding** characters to the right or **right-aligned** with **padding** characters to the left, respectively. The padding character is specified by the `fill` member function or the `setfill` parameterized stream manipulator (which we discuss in Section 19.6.7). Figure 19.11 uses the `setw`, `left` and `right` manipulators to left align and right align integer data in a field—we wrap each field in quotes so you can see the leading and trailing space in the field.

```
1  // fig19_11.cpp
2  // Left and right alignment with stream manipulators left and right.
3  #include <iomanip>
4  #include <iostream>
5
6  int main() {
7     int x{12345};
8
9     // display x right aligned (default)
10    std::cout << "Default is right aligned:\n\""
11       << std::setw(10) << x << "\"";
12
13    // use left manipulator to display x left aligned
14    std::cout << "\n\nUse left to left align x:\n\""
15       << std::left << std::setw(10) << x << "\"";
16
17    // use right manipulator to display x right aligned
18    std::cout << "\n\nUse right to right align x:\n\""
19       << std::right << std::setw(10) << x << "\"\n";
20 }
```

```
Default is right aligned:
"     12345"

Use left to left align x:
"12345     "

Use right to right align x:
"     12345"
```

Fig. 19.11 | Left and right alignment with stream manipulators `left` and `right`.

Stream manipulator `internal` (Fig. 19.12; line 8) indicates that a number's **sign** should be **left-aligned** within a field, the number's magnitude should be **right-aligned** and

intervening spaces should be padded with the **fill character**. When using stream manipulator `showbase`, the **base** is left aligned. The `showpos` manipulator (line 8) forces the plus sign to print. To reset the `showpos` setting, output the stream manipulator `noshowpos`.

```
1  // fig19_12.cpp
2  // Printing an integer with internal spacing and plus sign.
3  #include <iomanip>
4  #include <iostream>
5
6  int main() {
7     // display value with internal spacing and plus sign
8     std::cout << std::internal << std::showpos
9        << std::setw(10) << 123 << "\n";
10 }
```

```
+       123
```

Fig. 19.12 | Printing an integer with internal spacing and plus sign.

19.6.7 Padding (`fill`, `setfill`)

The `fill` member function specifies the **fill character** to use with aligned fields and returns the previous fill character. Spaces are used for padding by default. The `setfill` manipulator also sets the **padding character**. Figure 19.13 demonstrates `fill` (line 29) and `setfill` (lines 33 and 38) to set the fill character.

```
1  // fig19_13.cpp
2  // Using member function fill and stream manipulator setfill to change
3  // the padding character for fields larger than the printed value.
4  #include <iomanip>
5  #include <iostream>
6
7  int main() {
8     int x{10000};
9
10    // display x
11    std::cout << x << " printed as int right and left aligned\n"
12       << "and as hex with internal alignment.\n"
13       << "Using the default pad character (space):\n";
14
15    // display x
16    std::cout << std::setw(10) << x << "\n";
17
18    // display x with left alignment
19    std::cout << std::left << std::setw(10) << x << "\n";
20
```

Fig. 19.13 | Using member function `fill` and stream manipulator `setfill` to change the padding character for fields larger than the printed values. (Part 1 of 2.)

```
21      // display x with base as hex with internal alignment
22      std::cout << std::showbase << std::internal << std::setw(10)
23         << std::hex << x << "\n\n";
24
25      std::cout << "Using various padding characters:\n";
26
27      // display x using padded characters (right alignment)
28      std::cout << std::right;
29      std::cout.fill('*');
30      std::cout << std::setw(10) << std::dec << x << "\n";
31
32      // display x using padded characters (left alignment)
33      std::cout << std::left << std::setw(10) << std::setfill('%')
34         << x << "\n";
35
36      // display x using padded characters (internal alignment)
37      std::cout << std::internal << std::setw(10)
38         << std::setfill('^') << std::hex << x << "\n";
39   }
```

```
10000 printed as int right and left aligned
and as hex with internal alignment.
Using the default pad character (space):
     10000
10000
0x    2710

Using various padding characters:
*****10000
10000%%%%%
0x^^^^2710
```

Fig. 19.13 | Using member function `fill` and stream manipulator `setfill` to change the padding character for fields larger than the printed values. (Part 2 of 2.)

19.6.8 Integral Stream Base (dec, oct, hex, showbase)

C++ provides stream manipulators **dec**, **oct** and **hex** to specify that integers should display as decimal, hexadecimal and octal values, respectively. Integers display in decimal (base 10) by default. With stream extraction, integers prefixed with `0` are treated as octal values, integers prefixed with `0x` or `0X` are treated as hexadecimal values, and all other integers are treated as decimal values. Once you specify a stream's base, it processes all integers using that base until you specify a different one or until the program terminates.

Stream manipulator **showbase** causes octal numbers to be output with a leading `0` and hexadecimal numbers with either a leading `0x` or a leading `0X`—Section 19.6.10 shows that stream manipulator `uppercase` determines which option is chosen for hexadecimal values. Figure 19.14 demonstrates `showbase`. To reset the `showbase` setting, insert the stream manipulator **noshowbase** in the stream.

```
1  // fig19_14.cpp
2  // Stream manipulator showbase.
3  #include <iostream>
4
5  int main() {
6     int x{100};
7
8     // use showbase to show number base
9     std::cout << "Printing octal and hexadecimal values with showbase:\n"
10       << std::showbase;
11
12    std::cout << x << "\n"; // print decimal value
13    std::cout << std::oct << x << "\n"; // print octal value
14    std::cout << std::hex << x << "\n"; // print hexadecimal value
15 }
```

```
Printing octal and hexadecimal values with showbase:
100
0144
0x64
```

Fig. 19.14 | Stream manipulator `showbase`.

19.6.9 Floating-Point Numbers; Scientific and Fixed Notation (`scientific`, `fixed`)

When you display a floating-point number without specifying its format, its value determines the output format. Some numbers display in scientific notation and others in fixed-point notation. The sticky stream manipulators `scientific` and `fixed` control the output format of floating-point numbers:

- `scientific` forces a floating-point number to display in scientific format.
- `fixed` forces a floating-point number to display in fixed-point notation with a specific number of digits to the right of the decimal point, as specified by member function `precision` or stream manipulator `setprecision`.

Figure 19.15 displays floating-point numbers in fixed and scientific formats using stream manipulators `scientific` (line 15) and `fixed` (line 19). The exponent format in scientific notation might vary among compilers.

```
1  // fig19_15.cpp
2  // Floating-point values displayed in system default,
3  // scientific and fixed formats.
4  #include <iostream>
5
6  int main() {
7     double x{0.001234567};
8     double y{1.946e9};
9
```

Fig. 19.15 | Floating-point values displayed in system default, scientific and fixed formats. (Part 1 of 2.)

```
10      // display x and y in default format
11      std::cout << "Displayed in default format:\n" << x << '\t' << y;
12
13      // display x and y in scientific format
14      std::cout << "\n\nDisplayed in scientific format:\n"
15         << std::scientific << x << '\t' << y;
16
17      // display x and y in fixed format
18      std::cout << "\n\nDisplayed in fixed format:\n"
19         << std::fixed << x << '\t' << y << "\n";
20   }
```

```
Displayed in default format:
0.00123457      1.946e+09

Displayed in scientific format:
1.234567e-03    1.946000e+09

Displayed in fixed format:
0.001235        1946000000.000000
```

Fig. 19.15 | Floating-point values displayed in system default, scientific and fixed formats. (Part 2 of 2.)

19.6.10 Uppercase/Lowercase Control (uppercase)

Stream manipulator **uppercase** outputs an uppercase `X` with hexadecimal-integer values or an uppercase `E` with scientific-notation floating-point values (Fig. 19.16; line 11). Using `uppercase` also displays the hexadecimal digits `A`–`F` in uppercase. These appear in lowercase by default. To reset the `uppercase` setting, output **nouppercase**.

```
1   // fig19_16.cpp
2   // Stream manipulator uppercase.
3   #include <iostream>
4
5   int main() {
6      std::cout << "Printing uppercase letters in scientific\n"
7         << "notation exponents and hexadecimal values:\n";
8
9      // use std::uppercase to display uppercase letters; use std::hex and
10     // std::showbase to display hexadecimal value and its base
11     std::cout << std::uppercase << 4.345e10 << "\n"
12        << std::hex << std::showbase << 123456789 << "\n";
13  }
```

```
Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+10
0X75BCD15
```

Fig. 19.16 | Stream manipulator `uppercase`.

19.6.11 Specifying Boolean Format (boolalpha)

C++ `bool` values may be `false` or `true`. Recall that `0` also indicates `false`, and any nonzero value indicates `true`. A `bool` value displays as `0` or `1` by default. You can use stream manipulator **`boolalpha`** to set the output stream to display `bool` values as the strings `"true"` and `"false"` and stream manipulator **`noboolalpha`** to set the output stream back to displaying `bool` values as the integers `0` and `1`. Figure 19.17 demonstrates these stream manipulators. Line 9 displays `booleanValue` (which line 6 sets to `true`) as an integer. Line 13 uses `boolalpha` to display the `bool` value as a string. Lines 16–17 then change the `booleanValue` to `false` and use manipulator `noboolalpha`, so line 20 can display the `bool` value as an integer. Line 24 uses manipulator `boolalpha` to display the `bool` value as a string. Both `boolalpha` and `noboolalpha` are sticky settings.

```
1  // fig19_17.cpp
2  // Stream manipulators boolalpha and noboolalpha.
3  #include <iostream>
4
5  int main() {
6     bool booleanValue{true};
7
8     // display default true booleanValue
9     std::cout << "booleanValue is " << booleanValue;
10
11    // display booleanValue after using boolalpha
12    std::cout << "\nbooleanValue (after using boolalpha) is "
13       << std::boolalpha << booleanValue;
14
15    std::cout << "\n\nswitch booleanValue and use noboolalpha\n";
16    booleanValue = false; // change booleanValue
17    std::cout << std::noboolalpha; // use noboolalpha
18
19    // display default false booleanValue after using noboolalpha
20    std::cout << "\nbooleanValue is " << booleanValue;
21
22    // display booleanValue after using boolalpha again
23    std::cout << "\nbooleanValue (after using boolalpha) is "
24       << std::boolalpha << booleanValue << "\n";
25 }
```

```
booleanValue is 1
booleanValue (after using boolalpha) is true

switch booleanValue and use noboolalpha

booleanValue is 0
booleanValue (after using boolalpha) is false
```

Fig. 19.17 | Stream manipulators `boolalpha` and `noboolalpha`.

19.6.12 Setting and Resetting the Format State via Member Function `flags`

Throughout Section 19.7, we've used stream manipulators to change output format characteristics. How do you return an output stream's format to its previous state after changing its format? Member function **`flags`** without an argument returns the current **format state** settings as an **`fmtflags`** object. Member function `flags` with an `fmtflags` argument sets the format state as specified by the argument and returns the prior state settings. The initial settings of the value that `flags` returns might vary among compilers. Figure 19.18 uses member function `flags` to save the stream's original format state (line 16), then restore the original format settings (line 24). The capabilities for capturing the format state and restoring it are not required in C++20 text formatting. Each `std::format` call's formatting is used only in that call and does not affect any other `std::format` call, thus simplifying output formatting.

```
1  // fig19_18.cpp
2  // flags member function.
3  #include <format>
4  #include <iostream>
5
6  int main() {
7     int integerValue{1000};
8     double doubleValue{0.0947628};
9
10    // display flags value, int and double values (original format)
11    std::cout << std::format("flags value: {}\n", std::cout.flags())
12       << "int and double in original format:\n"
13       << integerValue << '\t' << doubleValue << "\n\n";
14
15    // save original format, then change the format
16    auto originalFormat{std::cout.flags()};
17    std::cout << std::showbase << std::oct << std::scientific;
18
19    // display flags value, int and double values (new format)
20    std::cout << std::format("flags value: {}\n", std::cout.flags())
21       << "int and double in new format:\n"
22       << integerValue << '\t' << doubleValue << "\n\n";
23
24    std::cout.flags(originalFormat); // restore format
25
26    // display flags value, int and double values (original format)
27    std::cout << std::format("flags value: {}\n", std::cout.flags())
28       << "int and double in original format:\n"
29       << integerValue << '\t' << doubleValue << "\n";
30 }
```

Fig. 19.18 | `flags` member function. (Part 1 of 2.)

```
flags value: 513
int and double in original format:
1000    0.0947628

flags value: 5129
int and double in new format:
01750   9.476280e-02

flags value: 513
int and double in original format:
1000    0.0947628
```

Fig. 19.18 | `flags` member function. (Part 2 of 2.)

19.7 Stream Error States

Each stream object contains a set of **state bits** representing the stream's state—sticky format settings, error indicators, etc. You can use this information to test, for example, whether an input was successful. These state bits are defined in class `ios_base`—the base class of the stream classes. Stream extraction sets the stream's **`failbit`** to `true` if the wrong type of data is input. Similarly, stream extraction sets the stream's **`badbit`** to `true` if the operation fails in an unrecoverable manner—for example, if a disk fails when a program is reading a file from that disk. Figure 19.19 shows how to use bits like `failbit` and `badbit` to determine a stream's state.[4] Online Appendix E discusses bits and bit manipulation in detail.

```
1  // fig19_19.cpp
2  // Testing error states.
3  #include <iostream>
4
5  int main() {
6     int integerValue{0};
7
8     // display results of cin functions
9     std::cout << std::boolalpha << "Before a bad input operation:"
10       << "\ncin.rdstate(): " << std::cin.rdstate()
11       << "\n    cin.eof(): " << std::cin.eof()
12       << "\n   cin.fail(): " << std::cin.fail()
13       << "\n    cin.bad(): " << std::cin.bad()
14       << "\n   cin.good(): " << std::cin.good()
15       << "\n\nExpects an integer, but enter a character: ";
16
17    std::cin >> integerValue; // enter character value
18
```

Fig. 19.19 | Testing error states. (Part 1 of 2.)

4. The actual values output by this program may vary among compilers.

```
19      // display results of cin functions after bad input
20      std::cout << "\nAfter a bad input operation:"
21         << "\ncin.rdstate(): " << std::cin.rdstate()
22         << "\n    cin.eof(): " << std::cin.eof()
23         << "\n   cin.fail(): " << std::cin.fail()
24         << "\n    cin.bad(): " << std::cin.bad()
25         << "\n   cin.good(): " << std::cin.good();
26
27      std::cin.clear(); // clear stream
28
29      // display results of cin functions after clearing cin
30      std::cout << "\n\nAfter cin.clear()"
31         << "\ncin.fail(): " << std::cin.fail()
32         << "\ncin.good(): " << std::cin.good() << "\n";
33   }
```

```
Before a bad input operation:
cin.rdstate(): 0
    cin.eof(): false
   cin.fail(): false
    cin.bad(): false
   cin.good(): true

Expects an integer, but enter a character: A

After a bad input operation:
cin.rdstate(): 2
    cin.eof(): false
   cin.fail(): true
    cin.bad(): false
   cin.good(): false

After cin.clear()
cin.fail(): false
cin.good(): true
```

Fig. 19.19 | Testing error states. (Part 2 of 2.)

Member Function eof

The program begins by displaying the stream's state before receiving any input from the user (lines 9–14). Line 11 uses member function `eof` to determine whether end-of-file has been encountered on the stream. In this case, the function returns `0` (`false`). The function checks the value of the stream's **`eofbit`** data member, which is set to `true` for an input stream after end-of-file is encountered after an attempt to extract data beyond the end of the stream.

Member Function `fail`

Line 12 uses the **`fail`** member function to determine whether a stream operation has failed. The function checks the value of the stream's `failbit` data member, which is set to `true` on a stream when a format error occurs and as a result no characters are input. For example, this might occur when you attempt to read a number but the user enters a string. In this case, the function returns `0` (`false`). When such an error occurs on input, the characters are not lost. Usually, recovering from such input errors is possible.

Member Function bad

Line 13 uses the **bad** member function to determine whether a stream operation failed. The function checks the value of the stream's `badbit` data member, which is set to `true` for a stream when an error occurs that results in the loss of data—such as reading from a file when the disk on which the file is stored fails. In this case, the function returns 0 (`false`). Generally, such serious failures are nonrecoverable.

Member Function good

Line 14 uses the **good** member function, which returns `true` if the `bad`, `fail` and `eof` functions would all return `false`. The function checks the stream's **goodbit**, which is set to `true` for a stream if none of the bits `eofbit`, `failbit` or `badbit` is set to `true` for the stream. In this case, the function returns `1` (`true`). I/O operations should be performed only on "good" streams.

Member Function rdstate

The **rdstate** member function (line 10) returns the stream's overall error state as an integer value. The function's return value could be tested, for example, by a `switch` statement that examines `eofbit`, `badbit`, `failbit` and `goodbit`. The preferred means of testing the state of a stream is to use member functions `eof`, `bad`, `fail` and `good`—using these functions does not require you to be familiar with particular status bits.

Causing an Error in the Input Stream and Redisplaying the Stream's State

Line 17 reads a value into an `int` variable. Enter a string rather than an `int` to force an error to occur in the input stream. At this point, the input fails and lines 20–25 once again call the streams's state functions. In this case, `fail` returns `1` (`true`) because the input failed. Function `rdstate` also returns a nonzero value (`true`) because at least one of the member functions `eof`, `bad` and `fail` returned `true`. Once an error occurs in the stream, function `good` returns `0` (`false`).

Clearing the Error State, So You May Continue Using the Stream

After an error occurs, you can no longer use the stream until you reset its error state. The **clear** member function (line 27) is used to restore a stream's state to "good" so that I/O may proceed on that stream. Lines 30–32 show that `fail` returns `0` (`false`) and `good` returns `1` (`true`), so the input stream can be used again.

The default argument for `clear` is `goodbit`, so the statement

```
std::cin.clear();
```

clears `cin` and sets `goodbit` for the stream. The statement

```
std::cin.clear(ios::failbit)
```

sets the `failbit`. You might want to do this when performing input on `cin` with a user-defined type and encountering a problem. The name `clear` might seem inappropriate in this context, but it's correct.

Overloaded Operators ! and bool

Overloaded operators can be used to test a stream's state in conditions. The `operator!` member function—inherited into the stream classes from class `basic_ios`—returns `true` if the `badbit`, the `failbit` or both are `true`. The `operator bool` member function (added

in C++11) returns `false` if the `badbit` is `true`, the `failbit` is `true` or both are `true`. These functions are useful in I/O processing when a `true`/`false` condition is being tested under the control of a selection statement or iteration statement. For example, you could use an `if` statement of the form 11

```
if (!std::cin) {
   // process invalid input stream
}
```

to execute code if `cin`'s stream is invalid due to a failed input. Similarly, you've already seen a `while` condition of the form

```
while (std::cin >> variableName) {
   // process valid input
}
```

which enables the loop to execute as long as each input operation is successful and terminates the loop if an input fails or the end-of-file indicator is encountered.

19.8 Tying an Output Stream to an Input Stream

Interactive command-line applications generally use an `istream` for input and an `ostream` for output. When a prompting message appears on the screen, the user responds by entering the appropriate data. Obviously, the prompt needs to appear before the input operation proceeds. With output buffering, outputs appear only

- when the buffer fills
- when outputs are flushed explicitly by the program or
- automatically at the end of the program.

C++ provides member function **`tie`** to synchronize (i.e., "tie together") an `istream` and an `ostream` to ensure that outputs happen (that is, they are flushed) before subsequent inputs. The call

```
std::cin.tie(&cout);
```

ties `cout` (an `ostream`) to `cin` (an `istream`). This particular call is redundant because C++ performs this operation automatically for the standard output and input streams. However, you might use this with other input/output stream pairs. To untie an input stream, `inputStream`, from an output stream, use the call

```
inputStream.tie(0);
```

19.9 C++20 Text Formatting 20

As we mentioned in Chapter 3, C++20 provides powerful string-formatting capabilities via the **`format` function** (in header **`<format>`**). These capabilities greatly simplify formatting by using a concise syntax that's based on the Python programming language's text formatting. As you've seen in this chapter, pre-C++20 output formatting is quite verbose. C++20 text-formatting capabilities are more concise and more powerful.

We presented several new C++20 text-formatting features throughout the book using the open-source `{fmt}` library's `fmt::format` function because our preferred compilers did not yet support the C++20 `<format>` header and `std::format` function. As we wrote this 20

online chapter, Microsoft Visual C++ added support for these, so we used the `<format>` header and `std::format` function for this section.

19.9.1 C++20 `std::format` Presentation Types

When you specify a placeholder for a value in a format string, the `std::format` function assumes the value should be displayed as a string unless you specify another type. In some cases, the type is required—for example, if you want to specify precision for a floating-point number or change the base in which to display an integer. In these cases, you can specify the **presentation type** in each placeholder. Figure 19.20 shows the various presentation types.

```
// fig19_20.cpp
// C++20 text-formatting presentation types.
#include <format>
#include <iostream>

int main() {
   // floating-point presentation types
   std::cout << "Display 17.489 with default, .1 and .2 precisions:\n"
      << std::format("f: {0:f}\n.1f: {0:.1f}\n.2f: {0:.2f}\n\n", 17.489);

   std::cout << "Display 10000000000000000.0 with f, e, g and a\n"
      << std::format("f: {0:f}\ne: {0:e}\ng: {0:g}\na: {0:a}\n\n",
            10000000000000000.0);

   // integer presentation types; # displays a base prefix
   std::cout << "Display 100 with d, #b, #o and #x:\n"
      << std::format(
            "d: {0:d}\n#b: {0:#b}\n#o: {0:#o}\n#x: {0:#x}\n\n", 100);

   // character presentation type
   std::cout << "Display 65 and 97 with c:\n"
      << std::format("{:c} {:c}\n\n", 65, 97);

   // string presentation type
   std::cout << "Display \"hello\" with s:\n"
      << std::format("{:s}\n", "hello");
}
```

```
Display 17.489 with default, .1 and .2 precisions:
f: 17.489000
.1f: 17.5
.2f: 17.49

Display 10000000000000000.0 with f, e, g and a
f: 10000000000000000.000000
e: 1.000000e+16
g: 1e+16
a: 1.1c37937e08p+53
```

Fig. 19.20 | C++20 text-formatting presentation types. (Part 1 of 2.)

```
Display 100 with d, #b, #o and #x:
d: 100
#b: 0b1100100
#o: 0144
#x: 0x64

Display 65 and 97 with c:
A a

Display "hello" with s:
hello
```

Fig. 19.20 | C++20 text-formatting presentation types. (Part 2 of 2.)

Floating-Point Values and C++20 Presentation Type `f` 20

You've used the **presentation type** `f` to format floating-point values. Formatting is **type-dependent**, so this presentation type is required to specify a floating-point number's precision. Line 9 uses the `f` presentation type to format the `double` value `17.489` in the default precision (6) and rounded to the tenths and hundredths positions. Function `std::format` uses presentation types to determine whether the other formatting options are allowed for a given type. For all the presentation types, their formatting options and the order in which the options must be specified, see

`https://en.cppreference.com/w/cpp/utility/format/formatter`

Indexing Arguments By Position

In line 9, note the `0` to the left of each format specifier's colon (:). You can reference the arguments after the format string **positionally** by their index numbers starting from index `0`. This allows you to

- **reference the same argument multiple times**, as we did three times in line 9, or
- **reference the arguments in any order**, which can be helpful when localizing applications for spoken languages that order words differently in sentences.

Floating-Point Values and C++20 Presentation Types `e`, `g` and `a` 20

You also can format floating-point values using the following presentation types, as shown in lines 12–13:

- **`e` or `E`**—These use **exponential (scientific) notation** to format floating-point values. The exponent is preceded by an `e` or `E` in the formatted string. The value `1.000000e+16` in this program's output is equivalent to

  1.000000×10^{16}

- **`g` or `G`**—These choose between fixed-point notation and exponential notation based on the value's magnitude. For exponential notation, `g` displays a lowercase `e`, and `G` displays an uppercase `E`.
- **`a` or `A`**—These format floating-point values in hexadecimal notation with lowercase letters (`a`) or uppercase letters (`A`), respectively.

20 **C++20 Integer Presentation Types**

Lines 17–18 display the `int` value `100` using various integer number systems:[5]

- **`d`**—Displays an integer in decimal (base 10) format.
- **`b` or `B`**—These display an integer in binary (base 2) format with a lowercase `b` or uppercase `B` when a binary value is displayed with its base (Section 19.9.3).
- **`o` presentation type**—Displays an integer in octal (base 8) format.
- **`x` or `X` presentation type**—These display an integer in hexadecimal (base 16) format with a lowercase or uppercase letters, respectively.

20 **C++20 Character Presentation Type**

The **`c` presentation type** formats an integer character code as the corresponding character, as shown in line 22.

20 **C++20 String Presentation Type**

When a placeholder does not specify a presentation type, the default is to format the corresponding value as a string. The **`s` presentation type** indicates that the corresponding value must specifically be a string, an expression that produces a string or a string literal, as in line 26.

20 **C++20 Locale-Specific Numeric and `bool` Formatting**

If your application requires locale-specific formatting of numeric or `bool` values, precede the integer or floating-point presentation type with `L`.

20 ## 19.9.2 C++20 `std::format` Field Widths and Alignment

Previously you used **field widths** to format text in a specified number of character positions. Figure 19.21 demonstrates field widths, default alignments and explicit alignments. We enclose each formatted value in brackets (`[]`) so you can better see the formatting results. By default, `std::format` **right-aligns numbers** and **left-aligns strings**, as demonstrated by line 8. For values with fewer characters than the field width, the remaining character positions are filled with spaces.[6] Values with more characters than the specified field width use as many character positions as they need.

```
1  // fig19_21.cpp
2  // C++20 text-formatting with field widths and alignment.
3  #include <format>
4  #include <iostream>
5
6  int main() {
```

Fig. 19.21 | C++20 text-formatting with field widths and alignment. (Part 1 of 2.)

5. See the online Appendix C, Number Systems for information about the binary, octal and hexadecimal number systems.

20 6. C++20 allows you to specify any fill character (other than `{` and `}`, which delimit placeholders) immediately to the right of the format specifier's colon (`:`).

```
7      std::cout << "Default alignment with field width 10:\n"
8         << std::format("[{:10d}]\n[{:10f}]\n[{:10}]\n\n", 27, 3.5, "hello");
9
10     std::cout << "Specifying left or right alignment in a field:\n"
11        << std::format("[{:<15d}]\n[{:<15f}]\n[{:>15}]\n\n",
12              27, 3.5, "hello");
13
14     std::cout << "Centering text in a field:\n"
15        << std::format("[{:^7d}]\n[{:^7.1f}]\n[{:^7}]\n", 27, 3.5, "hello");
16  }
```

```
Default alignment with field width 10:
[        27]
[  3.500000]
[hello     ]

Specifying left or right alignment in a field:
[27             ]
[3.500000       ]
[          hello]

Centering text in a field:
[  27   ]
[  3.5  ]
[ hello ]
```

Fig. 19.21 | C++20 text-formatting with field widths and alignment. (Part 2 of 2.)

Recall that you can specify left-alignment and right-alignment with < and >. Lines 11–12 **left-align** the numeric values `27` and `3.5` and **right-align** the string `"hello"` in fields of 15 characters

The I/O streams output formatting shown earlier in this chapter does not support **center-aligning text**, but `std::format` can do this conveniently with ^, as shown in line 15. Centering attempts to spread the unoccupied character positions equally to the left and right of the formatted value. `std::format` places the extra space to the right if an odd number of character positions remain, as you can see for the value `27`, which has two spaces to its left and three to its right.

19.9.3 C++20 `std::format` Numeric Formatting

Figure 19.22 demonstrates various C++20 numeric formatting capabilities. By default, negative numeric values display with a – sign. Sometimes it's desirable to force a + sign to display for a positive number. A + in the format specifier (line 8) indicates that the numeric value should always be preceded by a sign (+ or -). To fill the field's remaining characters with `0`s rather than spaces, place `0` before the field width and *after* the + if there is one, as in line 8's second format specifier. A space in the format specifier (as in line 11's second and third format specifiers) indicates that positive numbers should show a space character in the sign position. This is useful for aligning positive and negative values for display purposes. Note that the two values with a space in their format specifiers align. If a field width is specified, place the space before the field width. To precede a binary, octal or hexadecimal number with its base, use # in the format specifier as in line 14.

```
1  // fig19_22.cpp
2  // C++20 text-formatting numeric formatting options.
3  #include <format>
4  #include <iostream>
5
6  int main() {
7     std::cout << "Displaying signs and padding with leading 0s:\n"
8        << std::format("[{0:+10d}]\n[{0:+010d}]\n\n", 27);
9
10    std::cout << "Displaying a space before a positive value:\n"
11       << std::format("{0:d}\n{0: d}\n{1: d}\n\n", 27, -27);
12
13    std::cout << "Displaying a base indicator before a number:\n"
14       << std::format("{0:d}\n{0:#b}\n{0:#o}\n{0:#x}\n", 100);
15 }
```

```
Displaying signs and padding with leading 0s:
[       +27]
[+000000027]

Displaying a space before a positive value:
27
 27
-27

Displaying a base indicator before a number:
100
0b1100100
0144
0x64
```

Fig. 19.22 | C++20 text-formatting numeric formatting options.

19.9.4 C++20 `std::format` Field Width and Precision Placeholders

You can programmatically specify field widths and precisions using **nested placeholders** in a format specifier. Figure 19.23 displays the `double` value `123.456` in a field of `8` characters with precisions of `0`–`4`. In line 13, the argument `value` is the number to format. In the format specifier `"{:{}.{}f}"`, the nested placeholders to the right of the colon (`:`) are replaced left-to-right by the values of the arguments `width` and `precision`, respectively.

```
1  // fig19_23.cpp
2  // C++20 text-formatting field width and precision placeholders.
3  #include <format>
4  #include <iostream>
5
6  int main() {
7     std::cout << "Demonstrating field width and precision placeholders:\n";
8
9     double value{123.456};
10    int width{8};
```

Fig. 19.23 | C++20 text-formatting field width and precision placeholders.

```
11
12      for (int precision{0}; precision < 5; ++precision) {
13         std::cout << std::format("{:{}.{}f}\n", value, width, precision);
14      }
15   }
```

```
Demonstrating field width and precision placeholders:
     123
   123.5
  123.46
 123.456
123.4560
```

Fig. 19.23 | C++20 text-formatting field width and precision placeholders.

19.10 Wrap-Up

This chapter showed C++ input/output formatting with streams. You learned about the stream-I/O classes and predefined objects. We discussed `ostream`'s formatted and unformatted output capabilities performed by the `put` and `write` functions. You learned about `istream`'s formatted and unformatted input capabilities performed by the `eof`, `get`, `getline`, `peek`, `putback`, `ignore` and `read` functions. We discussed stream manipulators and member functions that perform formatting tasks:

- `dec`, `oct`, `hex` and `setbase` for displaying integers
- `precision` and `setprecision` for controlling floating-point precision
- and `width` and `setw` for setting field width.

You also learned additional formatting with `iostream` manipulators and member functions:

- `showpoint` for displaying decimal point and trailing zeros
- `left`, `right` and `internal` for alignment
- `fill` and `setfill` for padding
- `scientific` and `fixed` for displaying floating-point numbers in scientific and fixed notation
- `uppercase` for uppercase/lowercase control
- `boolalpha` for specifying Boolean format
- and `flags` and `fmtflags` for resetting the format state.

You'll encounter many of the preceding capabilities in legacy C++ code.

Finally, we presented many of C++20's more concise and convenient text-formatting capabilities, including presentation types to specify data types to format, positional arguments, field widths, alignment, numeric formatting and using placeholders to specify field widths and precisions.