Formatting Stream Output

You can change the way an output stream formats data using *stream manipulators*. You apply a stream manipulator by inserting it into the stream using its `<<` operator together with the data itself. With the `setprecision()` manipulator of the `<iomanip>` module, for instance, you can tweak the number of decimal digits the stream uses to format floating-point numbers. Here is an example.

```
std::cout << "Pond diameter required for " << fish_count << " fish is "
          << std::setprecision(2)            // Use two significant digits
          << pond_diameter << " feet.\n";    // Output value is 8.7
```

The standard `<ios>` and `<iomanip>` modules define many more stream manipulators. Examples include `std::hex` (produces hexadecimal numbers), `std::scientific` (enables exponent notation for floating-point numbers), and `std::setw()` (used to format tabular data). We do not discuss stream manipulators in detail here, however, because in this book we will use `std::format()` instead. The reason is that, compared to stream manipulators, this C++20 function is slightly more powerful, tends to result in more compact and readable code, and is often faster in execution. You can consult a Standard Library reference to find out more about stream manipulators. Doing so should be a walk in the park once you know `std::format()`, though, as formatting with manipulators is based on the exact same concepts as formatting with `std::format()` (width, precision, fill characters, and so on).

String Formatting with std::format()

After importing C++20's `<format>` module, you can replace the output statement of `Ex2_03A` with the following one (note that in `Ex2_03A` the type of `fish_count` was changed from `double` to `unsigned int`):

```
std::cout << std::format("Pond diameter required for {} fish is {} feet.\n",
                         fish_count, pond_diameter);
```

The first argument to `std::format()` is always the *format string*. This string contains any number of *replacement fields*, each surrounded with a pair of curly braces, `{}`. The format string is followed by zero or more additional *arguments*, generally one per replacement field. In our example there are two: `fish_count` and `pond_diameter`. The result of `std::format()` then is a copy of the format string where each field is replaced with a textual representation of one of these arguments.

In our initial example both replacement fields are empty. This means that `std::format()` will match them with the other two arguments, `fish_count` and `pond_diameter`, in left-to-right order,and that it will use its default formatting rules to convert these values into text. Here is the expected output (you can find the complete program in `Ex2_03B.cpp`*):*

```
Pond diameter required for 20 fish is 8.740387444736633 feet.
```

A good start. Only: we seem to have made matters worse. We're now digging ponds with sub-femptometer precision—a precision that is about a million times smaller than your average atom. The reason is that the default precision of `std::format()` is such that it ensures so-called *lossless round-trips*. This means that if you convert any formatted number back into a value of the same type, you by default obtain the exact same value as the one you started with. For a `double` this can result in strings with up to 16 decimal digits[3], as in our example.

[3] Not all `double`s will be formatted using 16 digits, though. The default formatting of the `double` value `1.5`, for instance, would just be `"1.5"`.

But of course you can override the default formatting. You do so by adding a *format specifier* between the curly braces of a replacement field. We discuss this in the next subsection. After that, we'll explain you how to override the default left-to-right order, and how to output the same value more than once.

Format Specifiers

A format specifier is a seemingly cryptic sequence of numbers and characters, introduced by a colon, telling `std::format()` how you would like the corresponding data to be formatted. To tune floating-point precision, for instance, you add, after the mandatory colon, a dot followed by an integer number:

```
std::cout << std::format("Pond diameter required for {} fish is {:.2} feet.\n",
                         fish_count, pond_diameter);
```

By default, this integer specifies the *total number of significant digits* (2 in our example), counting digits both before and after the decimal point. The result thus becomes (see `Ex2_03C.cpp`):

```
Pond diameter required for 20 fish is 8.7 feet.
```

You can instead make the precision specify the number of digits *after* the decimal point—the *number of decimal places* in other words—by enabling so-called "fixed-point" formatting of floating-point numbers. You do so by appending the letter `f` to the format specifier. If you replace `{:.2}` with `{:.2f}` in our running example, it produces the following output:

```
Pond diameter required for 20 fish is 8.74 feet.
```

■ **Note** You obtain the equivalent with stream manipulators by inserting '`<< std::precision(2) << std::fixed`' into an output stream. See how much more compact '`{:.2f}`' is?

Here is the (slightly simplified) general form of the format specifiers for fields of fundamental and string types:

```
[[fill]align][sign][#][0][width][.precision][type]
```

The square brackets are for illustration purposes only and mark optional *formatting options*. Not all options are applicable to all field types. `precision`, for instance, is only applicable to floating-point numbers (as you already know) and strings (where it defines how many characters will be used from the string).

■ **Caution** If you specify an unsupported formatting option, or if you make any syntactical mistake in one of the format specifiers, `std::format()` will fail. To report this failure, `std::format()` raises what is known as an *exception*. If such an exception occurs, and there is no error handling code in place; your entire program instantly terminates with an error. You can give this a try by replacing the format string in `Ex2_03C.cpp` with the following (we added a precision to the first format specifier):

```
std::cout << std::format("Pond diameter required for {:.2} fish is {:.2} feet.\n",
                         fish_count, pond_diameter);
```

Because `fish_count` is an integer, you are not allowed to specify a precision for that field. And so this invocation of `std::format()`, and by extension your entire program, will fail when executed.

Even though we only explain how to handle exceptions in Chapter 16, you need a means to debug failing `std::format()` statements. After all: you will be using `std::format()` in examples and exercises long before Chapter 16, and finding out which tiny little mistake you made in a format specifier can be tricky at times. Hence, the following tip:

■ **Tip** To debug a failing `std::format()` expression, you can put the corresponding statement inside a so-called `try-catch` block as follows (this code is available in `Ex2_03D.cpp`):

```
try
{
  std::cout << std::format("Pond diameter required for {:.2} fish is {:.2} feet.\n",
                           fish_count, pond_diameter);
}
catch (const std::format_error& error)
{
  std::cout << error.what();  // Outputs "precision not allowed for this argument type"
}
```

The program will now no longer fail and instead will output a diagnostic message that should aid you in fixing the buggy format specifier. This `try-catch` snippet uses several language elements you do not know yet (references, exceptions, etc.), but you can easily copy-paste it from `Ex2_03D.cpp` whenever you need it.

Formatting Tabular Data

The following formatting options (highlighted in black) allow you to control the width and alignment of each field. We regularly use them in this book to output text that resembles a table.

```
[[fill]align][sign][#][0][width][.precision][type]
```

`width` is a positive integer that defines the *minimum field width*. If needed, extra characters are inserted into the formatted field to reach this minimum width. Which characters are inserted and where depends both on the field's type and which other formatting options are present:

- For numeric fields, if the `width` option is preceded with `0` (zero), extra `0` characters are inserted before the number's digits, but after any sign character (`+` or `-`) or prefix sequence (such as `0x` for hexadecimal numbers: see later).
- Otherwise, the so-called *fill character* is inserted. The default fill character is a space, but you can override this with the `fill` formatting option. The `align` option determines where this fill character is inserted. A field can be *left-aligned* (`<`), *right-aligned* (`>`), or *centered* (`^`). The default alignment depends on the field's type.

You cannot specify the `fill` character without specifying an `align`ment as well. Note also that the `fill`, `align`, and `0` options have no effect unless you also specify a `width`.

Still with us? Or is your head starting to spin? We know: we just threw a lot of formatting options at you all at once. High time we made it all concrete with some code. We recommend that you take your time to analyze this next example (perhaps you can try to predict what the output will look like?), and to play with its format specifiers to see the effects of the different formatting options.

```
// Ex2_05.cpp
// The width, alignment, fill, and 0 formatting options of std::format()
import <iostream>;
import <format>;

int main()
{
  // Default alignment: right for numbers, left otherwise
  std::cout << std::format("{:7}|{:7}|{:7}|{:7}|{:7}\n", 1, -.2, "str", 'c', true);
  // Left and right alignment + custom fill character
  std::cout << std::format("{:*<7}|{:*<7}|{:*>7}|{:*>7}|{:*>7}\n", 1,-.2,"str",'c',true);
  // Centered alignment + 0 formatting option for numbers
  std::cout << std::format("{:^07}|{:^07}|{:^7}|{:^7}|{:^7}\n", 1, -.2, "str", 'c', true);
}
```

Because we use the same `width`, `7`, for all fields, the result resembles a table. More specifically, the result looks as follows:

```
      1|   -0.2|str    |c      |true
1******|-0.2***|****str|******c|***true
0000001|-0000.2|  str  |   c   | true
```

From the first line, you see that the default alignment of fields is not always the same. By default numeric fields align to the right, whereas string, character, and Boolean fields align to the left.

For the second line, we mirrored all default alignments using the `<` and `>` alignment options. We also set the fill character of all fields to *.

The third line showcases two more options: centered alignment (`^`) and the special `0` filling option for numeric fields (`0` may only be used for numeric fields).

Formatting Numbers

These four remaining options are mostly relevant to numeric fields:

```
[[fill]align][sign][#][0][width][.precision][type]
```

`precision` you know already from before, but there is much more to formatting numbers than that. Like in the previous section there is a lot to take in here, and we will make it more concrete with an example after.

- Fixed-point formatting (`f`; see earlier) is not the only supported `type` option for formatting floating-point numbers. Others are *scientific formatting* (`e`), *general formatting* (`g`), and even *hexadecimal formatting* (`a`).
- Integer fields are mostly equivalent to floating-point fields, except that the `precision` option is not allowed. Supported `type`s include *binary formatting* (`b`) and *hexadecimal formatting* (`x`).