Formatting of `chrono` Time Values

Document number: **P3148R0**
Date: 2024-02-14
Audience: Library Evolution Working Group
Reply to: **Alan Talbot**
cpp@alantalbot.com

History

During the 2023-12-19 LEWG telecon meeting, we reviewed two papers (P2945 and P3015) addressing some limitations in the C++20/23 support for formatting `chrono` library time values. There was strong consensus to preserve the existing formatting behaviors (which have been shipping in C++20 implementations), and equally strong consensus to request further work with the goal of addressing the concerns for C++26.

This paper is in response to that request. It addresses some of the issues raised by these papers and attempts to reconcile some of the differences. It also addresses some closely related issues that I believe are important.

Limitations in C++23

Fractional Seconds in Clock Times

The `%S` and `%T` conversion specifiers produce fractional seconds at the precision necessary to represent the underlying `time_point` (or 6 places if that isn't possible). This can be very useful but is not always desired. A means of controlling the precision (for instance, displaying whole seconds even though the underlying representation is in microseconds) is needed.

12-Hour Time

The `%I` and `%r` conversion specifiers produce hours which are zero-padded to two digits. This is not a presentation that is used in practice. A means of formatting the hours less than 10 as one digit is needed.

Furthermore, the `%r` conversion specifier suffixes the time with a space followed by either "AM" or "PM". While this format *is* used in Western language applications, many other variations can be found in the wild (some of which are more common). For example:

- The space may or may not be used.
- The spelling may be AM or A (PM or P).
- The case may be upper or lower.
- The distinction may be made using font alone—bold face for PM is typical.

While attempting to directly support all of these is almost certainly not necessary (see Appropriate Scope), a way to get 12-hour time without the suffix is needed so that other formats can be easily constructed.

There are no direct equivalents to `%T` and `%R` for 12-hour time. While they are not strictly necessary, this seems like an oversight given the very large number of people who use 12-hour time.

Fractional Durations

The `%H` and `%M` conversion specifiers produce whole hours and minutes but do not allow any way to get fractional hours or minutes. The `%S` conversion specifier produces fractional seconds but with no control over the precision. Fractional times are very common in many domains (e.g. hours and 10ths in a timesheet, or seconds and 100ths in a ski race report) so a way to produce them is needed.

All of these specifiers left zero pad numbers less than 10. This is needed to support clock time, but is never acceptable in other contexts, so a way to suppress it is needed.

The Standard is silent about modulus in time formatting, but it is assumed in all cases. There is no way to produce 24 or more hours (fractional or otherwise). The same issue exists with 60 or more minutes or seconds. These are all needed for non-clock applications. *Note: At least one major library produces total hours for `%H` (not modulo 24) when applied to durations.*

Seconds Since Epoch

As explained in P2945, all the other well-known time and date formatting systems provide a conversion specifier to extract the number of seconds since the clock's epoch (and they all use `%s`). This strong agreement in existing practice suggests that this feature is needed.

Workarounds

It is important to point out that all of these limitations can be circumvented in one way or another in C++23, either by converting to a `time_point` or `duration` based on a different ratio, or by converting to a built-in numeric type and formatting that using existing facilities. (In the case of 12-hour time, this would also require some math.) These manipulations are not particularly complex, nor are they likely to be costly at runtime, but I do not feel that it is appropriate to have to manipulate values mathematically to produce simple, widely used formatting results. I believe doing so is morally equivalent to writing:

```
println("{}", int(numbers::pi * 100000) / 100000.);
```

This is very outdated and error-prone at best, but fortunately it can be replaced by:

```
println("{:.5f}", numbers::pi);
```

Furthermore, the formatting facilities are meant to be accessible to beginning C++ users, while the mathematical intricacies of chrono types are (appropriately) not.

There are more technical and perhaps even more compelling reasons why formatting control is a better solution than value manipulation. These are discussed in P2945.

Appropriate Scope

An important question to ask about any additions to the Standard Library is: what should we try to support and what is best left to the user or the larger C++ community? In this case we have a very flexible and powerful time and date facility that we have decided is appropriate to embrace as part of the Standard Library (a decision with which I entirely agree). Given that, it seems only fair to the users of that library that we provide reasonably complete integration with other overlapping facilities of the Library such as formatting, especially as both time and date *and* output formatting are features that a very new C++ programmer is likely to want to use.

Conversely, attempting to support all the many possible formats for the 12-hour time AM/PM indication is neither necessary nor practical. In this case it is better to make it easy for the user to add whatever adornments are required. (See Examples for an example.)

Suggested Solutions

Modulus vs. Total

The `%i` conversion specifier produces the total (not modulo 24) fractional hours of the duration. The `%f` conversion specifier produces the total (not modulo 60) fractional minutes of the duration. The `%s` conversion specifier produces the total (not modulo 60) fractional seconds of the duration. Note that for `time_point`s, `%s` will be seconds since the clock's epoch, thus matching other time formatting facilities (see P2945).

Precision

The optional precision specification affects the `%f %H %i %M %s %S` conversion specifiers for all duration types (not just for floating point durations as in C++23). It will truncate the value of the lowest order place to the precision as if by `floor`. For `%H` and `%M` the presence of a non-zero precision will produce fractional hours or minutes.

Note: The method of truncating numbers to a precision (floor, ceiling, or round) has to be specified (I have chosen floor arbitrarily), but does not have to be user-controllable. It seems very unlikely that users will be concerned about the value of the first discarded digit. For environments where that level of control is needed, the aforementioned workarounds provide flexible solutions.

For the `%i %f %s %S` conversion specifiers, if the precision specification is not present, and the precision of the input cannot be exactly represented with an integer, then the format is a decimal floating-point number with a fixed format and a precision matching that of the precision of the input (or a precision of 6 places if the conversion to floating-point cannot be made within 18 fractional digits). *Note: This is a slight modification of the text in tab:time.format.spec for `%S`.*

12-hour Time

The `%J` and `%K` conversion specifiers provide 12-hour versions of `%T` and `%R` respectively, with no zero padding of hours:

`%J` has the same effect as `%EI:%M:%S`
`%K` has the same effect as `%EI:%M`

The optional precision specification affects these conversion specifiers as expected:

`.0%J` has the same effect as `.0%EI:%M:%S`

Zero Padding

The `E` modifier may be applied to the `%H %I %M %r %R %S %T` conversion specifiers. It will suppress leading zeros of the highest order place. Specifically:

`%Er` has the same effect as `%EI:%M:%S %p` (in the C local)
`%ER` has the same effect as `%EH:%M`
`%ET` has the same effect as `%EH:%M:%S`

Examples

Here are several examples to illustrate this design. A white background indicates existing C++23 behavior; a green background indicates proposed behavior.

Assume a `time_point tp` with a value of 13:23:45.678 (in milliseconds), the clock epoch starts today (to keep things simple), and `h = floor<hours>(tp - floor<days>(tp))`.

Format String	Output	Parameters
{:%T}	13:23:45.678	tp
{:.0%T}	13:23:45	tp
{:%H:%M:%S}	13:23:45.678	tp
{:.0%H:%M:%S}	13:23:45	tp
{:.1%H:%M:%S}	13:23:45.6	tp
{:.2%H:%M}	13:23.76	tp
{:%I:%M:%S}	01:23:45.678	tp
{:.0%EI:%M:%S}	1:23:45	tp
{:%r}	01:23:45 PM	tp
{:%J}	1:23:45	tp
{:%R}	13:23	tp
{:%K}	1:23	tp
{:%K}{}	1:23p	tp, is_am(h) ? 'a' : 'p'
{:%s}	48225.678	tp

Assume a `duration d` with a value of 9245678 milliseconds:

Format String	Output	Parameters
{:%H}	02	d
{:.2%EH}	2.56	d
{:.5%i}	2.56824	d
{:%M}	34	d
{:.2%M}	34.09	d
{:.0%f}	154	d
{:%S}	05.678	d
{:.1%S}	05.6	d
{:.1%ES}	5.6	d
{:%s}	9245.678	d
{:.0%s}	9245	d
{}	9245678ms	d
{}	9245678	d.count()

Alternative Designs

P2945 makes the point that using the existing precision specifier can put the precision confusingly far from its intended target. For example, the precision here applies to seconds, not hours:

```
.0%H:%M:%S
```

The suggested alternative is to put the precision specifier inside the format specifier thus:

```
%H:%M:%.0S
```

While I understand the motivation for this design, I believe that the disadvantages outweigh the benefits, and that it won't be an issue in practice anyway. My reasoning is as follows:

1. The precision specifier at the start of the format string is existing behavior, not just for times but for built-in types. I believe it would be confusing to prohibit it there for times.
2. However, if we don't prohibit it, what would using it do exactly? For example, what would it mean to say: `.1%H:%M:%.2S`
3. We also end up with things like: `%.3H:%.2M:%.1S` What does this do? I cannot think of any use cases for more than one precision in any one time format string.
4. I believe this will rarely come up in real use. The use of precision on hours and minutes is very unlikely to be combined with colon notation. For precision on seconds, no one is likely to use `.1%H:%M:%S` when `.1%T` does the same thing.

Another option suggested by P2945 is to add a trailing precision specifier like so:

```
%H:%M:%S%.0f
```

This design also has the advantage of putting the precision close to the specifier to which it applies, but all four of the points above still apply. Furthermore, it requires two extra characters (the surrounding `%` and `f`) to disambiguate it from regular text.

References

P2945R0: *Additional format specifiers for time_point.* Revzin.
P3015R0: *Rebuttal to Additional format specifiers for time_point.* Hinnant.

Acknowledgements

Thanks to Barry Revzin, Howard Hinnant and Victor Zverovich for their feedback on this proposal.