# Formatting of chrono Time Values

*Document number:* **P3148R1**
*Date:* 2025-01-13
*Audience:* Library Evolution Working Group
*Reply to:* **Alan Talbot**
cpp@alantalbot.com

## History

### R1

P3148R0 and P2945R1 were partially reviewed in Wrocław. There was considerable discussion resulting in consensus in favor of "the need for additional timepoint specifiers" and "the need for defining precision on timepoint specifiers", both "along the lines of what is described in [these papers]".

This revision proposes a different approach to the zero-padding problem and clarifies the design. There are also improvements to the discussion of the position and behavior of the precision specifier, a particular concern raised in Wrocław.

### R0

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.

## Abstract

The chrono library text formatting facility [time.format] has no convenient way to adjust the precision of fractional seconds (for integral timepoints and durations). It also lacks a few related features and has some limitations and problems in its handling of 12 hour time. This paper is a response to P2945 and P3015 that addresses these issues.

## 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 timepoint (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 ever 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* widely used in Western language applications, many other variations are also seen (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 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 zero pad numbers less than 10. This is needed to support clock time but is usually unacceptable in other contexts, so a way to avoid it is needed.

The Standard is silent about modulus in time formatting, but it is assumed for minutes and seconds by major implementations. Interestingly, it is not assumed for hours, so modulo 24 hours is not provided. Unbounded minutes and especially seconds are needed for non-clock applications, and modulo 24 hours are needed for multiple day timers.

**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`). Such 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 one way or another in C++23, either by converting to a timepoint 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 (quite appropriately) not.

There are more technical reasons why formatting control is a better solution than value manipulation. These are discussed in more detail in P2945, but the basic issue is that there are situations where the user does not have access to the chrono objects (or access is difficult), but *can* provide specifiers for formatting the chrono objects.

## Appropriate Scope

An important question to ask about any addition 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.

## Suggested Solutions

*New features have green backgrounds in the tables.*

### Hours

Hours are a special case because the existing `%H` is zero-padded but is not modulo 24. Hours that can run past 23 but are zero-padded below 10 are probably of very limited use, but that's what the existing specifier seems to do. The standard needs to clarify this and provide both modulo 24 and non-padded fractional hours. If padded fractional hours are needed, the fill feature can easily provide them.

| Specifier | Meaning | 29h + 15min | 3h | 16h |
|---|---|---|---|---|
| `%H` | Hours, zero-padded | `29` | `03` | `16` |
| `%K` | Hours modulo 24, zero-padded | `05` | `03` | `16` |
| `%k` | Hours, fractional | `29.25` | `3` | `16` |
| `%I` | Hours modulo 12, zero-padded | `05` | `03` | `04` |
| `%i` | Hours modulo 12 | `5` | `3` | `4` |

**Minutes**

Minutes lack a way to get fractions and remove the leading zero. Since it's easy to add the leading zero back in if you need it, and it's easy to remove any unwanted precision, only one additional specifier is needed.

| Specifier | Meaning | 15min+45s | 3min |
|---|---|---|---|
| `%M` | Minutes modulo 60, zero-padded | `15` | `03` |
| `%f` | Minutes, fractional | `15.75` | `3` |

**Seconds**

Seconds are just like minutes, but the `%S` specifier already provides fractional seconds. Since it's easy to remove any unwanted precision, only one additional specifier is needed. Note that for timepoints, `%s` will be seconds since the clock's epoch, thus matching the time formatting facilities in other languages (see P2945).

| Specifier | Meaning | 90s+500ms | 3s+500ms |
|---|---|---|---|
| `%S` | Seconds modulo 60, zero-padded, fractional | `30.500` | `03.500` |
| `%s` | Seconds, fractional | `90.500` | `3.500` |

**Composite Specifiers**

These are convenience specifiers which reduce typing (and are easier to remember). The proposed versions handle 12 hour time (correctly).

| Specifier | Equivalent | Meaning | 15h+25min+45s |
|---|---|---|---|
| `%T` | `%H:%M:%S` | 24 hour time with fractional seconds | `15:25:45` |
| `%N` | `%i:%M:%S` | 12 hour time with fractional seconds | `3:25:45` |
| `%R` | `%H:%M` | 24 hour time (no seconds) | `15:25` |
| `%P` | `%i:%M` | 12 hour time (no seconds) | `3:25` |
| `%r` | `:%I:%M:%S %p` | 24 hour time mod 12 w/ frac seconds & AM/PM | `03:25:45 PM` |
| `%l` | `:%i:%M:%S %p` | 12 hour time with fractional seconds and AM/PM | `3:25:45 PM` |

**Precision**

The optional precision specification affects any fractional conversion specifier, regardless of where it appears in the format specification. If there is more than one fractional conversion specifier in the format specification, all are affected (although I can think of no use case where more than one fractional conversion specifier would appear in a single format specification). The C++23 behavior for floating point duration types is unchanged, but the prohibition against using a precision specification on non-floating point types is removed. The precision specification will truncate the value to the precision as if by `floor`.

*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 fractional 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 meant to match the behavior specified for `%S` in C++23.*

## Impact

My approach has been to make this a pure addition—no existing code would be changed (in recognition of Hyrum's Law). If LEWG feels that changing the meaning of some existing features is a possibility, then this proposal can be implemented with better and fewer letters and the library would be more consistent and easier to understand. My ideas for changing existing meanings are:

Change the behavior of `%I` to match the proposed `%i`. This is very unlikely to break code, since anyone using `%I` today has to check for the zero and remove it.

Change the behavior of `%r` to match the proposed `%l`. This is very unlikely to break code, since anyone using `%r` today has to check for the zero and remove it. Another alternative is to remove `%r` altogether (and get back another character). Even fixed it's not very useful since it only offers one of many formatting conventions (see **12-Hour Time** above).

I noticed in passing that there are three redundant conversion specifiers: `%h`, `%n`, and `%t`. All of these characters would be useful, either for this proposal (`%h`) or for future enhancements.

## 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 a different precision specifier adjacent to the format specifier:

```
%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 is already confusing and annoying to prohibit it for integral times (as C++23 does). Adding another precision specifier in the same format specifier would make the confusion worse.
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 specifier which produces the fractional part of the seconds value. But since the latest version of that paper proposes a modifier on seconds to specify precision, this would seem to serve no purpose. It also suggests that a decimal be permitted within the specifier to include the decimal in the output, but that is not necessary since placing a decimal before the % would have the same effect.

## References

P2945R0/1: *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.