

# Comparing Two Designs for the **Time** Class

Douglas Wilhelm Harder

We have implemented the same abstraction—a duration of time—using two different internal representations.

The first design, `Time_hms`, stores separate components:

```
bool          is_negative_;
std::uint8_t  seconds_;
std::uint8_t  minutes_;
std::uint32_t hours_;
```

The second design, `Time_s`, stores the entire duration as a single signed number of seconds:

```
std::int64_t seconds_;
```

Both classes represent exactly the same kind of information and offer the same operations through corresponding public interfaces. The difference is entirely in how that information is stored internally.

## Human Representation versus Computer Representation

At first, the `Time_hms` representation seems more natural.

A human can more easily visualize

**300 hours**

than

**1080000 seconds**

Similarly, it is easier to understand that one must wait for approximately one hour than it does to be asked to wait for 3572 seconds. This does not mean that the computer should use the same representation.

The user of the class may provide hours, minutes, and seconds as input and expect hours, minutes, and seconds as output. That specifies the *interface*; it does not specify the internal representation.

This is an important principle of software design:

The representation that is most convenient for input and output need not be the representation that is most convenient for computation.

## Construction

Construction favors `Time_hms`. The input already consists of hours, minutes, and seconds, so these values can be stored almost directly.

For `Time_s`, the components must first be converted:

```
hours * 3600 + minutes * 60 + seconds
```

This is additional work, but it is a very simple calculation that is performed only when the object is constructed.

## Accessing Hours, Minutes, and Seconds

The accessors also favor `Time_hms`. Each requested component is already stored.

For `Time_s`, the components must be recovered using integer division and remainder.

Conceptually,

```
hours    = magnitude / 3600
minutes  = (magnitude / 60) % 60
seconds  = magnitude % 60
```

Again, the calculations are short and inexpensive.

## The Sign

A duration can be negative.

The `Time_hms` representation therefore requires an additional Boolean member to store the sign. It must also maintain the invariant that zero is never represented as negative zero.

The `Time_s` representation needs nothing extra. A negative duration is simply represented by a negative integer.

Thus the sign is already part of the representation.

## Negation

Negation is also simpler with `Time_s`. Negating a duration is equivalent to negating one integer.

With `Time_hms`, the magnitude remains unchanged and the separate sign must be changed. Zero must be treated carefully so that

`0:00:00`

does not become

`-0:00:00`

## Addition

Addition shows a much larger difference.

For `Time_hms`, suppose we add

`2:45:50`

`1:30:25`

We first add the seconds:

`50 + 25 = 75 seconds`

This must become 15 seconds with a carry of one minute.

We then add the minutes, including the carry:

`45 + 30 + 1 = 76 minutes`

This becomes 16 minutes with a carry of one hour.

Finally, the hours are added.

Even this assumes that both durations have the same sign. If their signs differ, we cannot simply add the components. We must determine which duration has the larger magnitude and subtract the smaller magnitude from the larger one. The sign of the result depends on which magnitude was larger.

For `Time_s`, none of this is necessary. Addition is simply the addition of two signed integers.

## Subtraction

Subtraction makes the difference even clearer.

With `Time_hms`, subtracting components may require borrowing. For example, when subtracting

```
5:10:20
2:35:45
```

we cannot subtract 45 seconds from 20 seconds directly. We must borrow one minute, converting 20 seconds into 80 seconds.

This reduces the available minutes. We may then discover that we also need to borrow one hour before subtracting the minutes.

Signs introduce additional cases.

For `Time_s`, subtraction is simply the subtraction of two signed integers.

## Increment and Decrement

Both classes support prefix and postfix increment and decrement. Each operation changes the represented duration by one second, including when crossing a minute, an hour, or zero.

For `Time_hms`, increment and decrement can reuse compound assignment with a duration of one second. The compound-assignment operators must then handle carrying, borrowing, and possible changes of sign.

For `Time_s`, increment and decrement directly increase or decrease the stored integer. Prefix operators return the modified object by reference; postfix operators return a copy of the original value.

## Comparison

Suppose we implement `operator<`.

With `Time_hms`, we must first consider the signs. Any negative duration is less than any non-negative duration.

If both are non-negative, we compare the hours. If the hours are equal, we compare the minutes. If those are also equal, we compare the seconds.

For two negative durations, the logic reverses: the duration with the larger magnitude is actually the smaller value.

Thus comparison requires several decisions.

With `Time_s`, the ordinary ordering of signed integers is already exactly the ordering of durations:

```
seconds_ < other.seconds_
```

No special treatment of hours, minutes, seconds, or signs is required.

## Output

Output is one place where `Time_hms` is more convenient. The hours, minutes, and seconds required for display are already stored.

The `Time_s` implementation must calculate these components before displaying a value such as

```
-37:15:42
```

This is a reasonable place to perform such a conversion. Output exists for the benefit of humans, so converting the computer's convenient representation into a human-friendly representation is expected.

## Storage and Range

On the system used for this example, both designs occupy eight bytes. The exact size of the component-based design can depend on member alignment and padding.

The largest positive duration that `Time_hms` can represent is

$$(2^{32} - 1) \text{ hours, } 59 \text{ minutes, } 59 \text{ seconds,}$$

which is approximately 490,000 years.

The `Time_s` design uses a single signed 64-bit integer. Its largest positive duration is therefore

$$2^{63} - 1 \text{ seconds,}$$

which is approximately 292 billion years.

Thus, despite using the same amount of storage on this system, `Time_s` has a much larger internal numerical range. The common public interface, however, returns hours as `std::uint32_t`. Our examples deliberately use ordinary durations that both classes can represent and display correctly.

## Summary of the Comparison

| Operation           | Time_hms                | Time_s                      |
|---------------------|-------------------------|-----------------------------|
| Construction        | Very simple             | Simple calculation          |
| Accessors           | Very simple             | Division/remainder          |
| Determine sign      | Separate field          | Integer sign                |
| Negation            | Change sign carefully   | Negate integer              |
| Addition            | Carries and sign cases  | Integer addition            |
| Subtraction         | Borrows and sign cases  | Integer subtraction         |
| Increment/decrement | Carries, borrows, signs | Integer increment/decrement |
| Comparison          | Multiple cases          | Integer comparison          |
| Output              | Very simple             | Division/remainder          |
| Storage             | 8 bytes                 | 8 bytes                     |
| Range               | Large                   | Vastly larger               |

## Conclusion

The `Time_hms` representation has one significant advantage: it resembles the way humans naturally think about durations. Consequently, construction, accessors, and output are particularly easy to understand.

The `Time_s` representation is superior for essentially every operation involving the duration itself. Addition, subtraction, negation, increment, decrement, and comparison all become ordinary integer operations. It also requires fewer invariants and provides a much greater internal range using the same amount of storage on our system.

The key lesson is that the requirements describe what the class must do, not how its data must be stored.

If the requirements say that hours, minutes, and seconds are supplied as input and produced as output, it may be tempting to store hours, minutes, and seconds internally; that is not required.

A good internal representation is one that makes the important operations simple. Conversion between the external representation and the internal representation can occur at the boundary of the class.

In this example, humans may prefer to think in hours, minutes, and seconds.

For this set of operations, the computer is better off thinking in seconds.