

Time Class

Software design description *v.2*

Douglas Wilhelm Harder

October 4, 2026

1 Purpose

This document describes the second of two implementations of the same duration abstraction. The class `Time_s` satisfies the accompanying requirements specification, just as the first class, `Time_hms`, does. Both provide the same kinds of constructors, accessors, comparisons, arithmetic operators, and output operations. The class names differ so that a demonstration program can create and compare objects of both types.

The first design uses an internal representation that closely resembles the way people describe durations: a sign together with hours, minutes, and seconds. This second design instead stores a duration as one signed number of seconds. The purpose is to illustrate how a representation that is less convenient for human interpretation can be considerably simpler for computation.

The requirements specification describes *what* the two classes must do. This software design description explains *how* the second class will do it. The public behavior must agree even though the private representations and algorithms differ.

2 Design overview

A `Time_s` object contains only one private data member:

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

This integer records the complete signed duration in seconds. Positive values represent positive durations, negative values represent negative durations, and zero represents a duration of zero. There is no separate Boolean sign and no stored hour or minute component.

For example, the durations `2:15:30` and `-2:15:30` are represented by `8130` and `-8130`, respectively. The duration `-0:00:01` is simply represented by `-1`.

DES-1.1 The entire duration shall be stored in a single `std::int64_t` member named `seconds_`.

DES-1.2 The sign of the duration shall be represented by the sign of `seconds_`. No separate sign member is required.

This design makes the meaning of addition, subtraction, and comparison immediately apparent. The trade-off is that whenever a client requests hours, minutes, or seconds, the class must calculate those components from the stored total.

3 Class invariant

An invariant describes the conditions that must hold for every valid object. The component-based implementation must maintain the ranges of the minute and second fields, the relationship between the components and the sign, and the rule prohibiting negative zero.

Here, the invariant is much simpler.

DES-2.1 The value of `seconds_` shall be the signed total number of seconds represented by the object.

DES-2.2 Zero shall be represented by `seconds_ == 0`. No separate negative-zero representation is possible.

DES-2.3 The stored value must be representable by `std::int64_t`. The simplified academic implementation assumes arithmetic remains within this range.

There are no fields that can contain an invalid value such as 75 minutes or 80 seconds. The stored value is already normalized, in the sense that there is exactly one integer for every representable duration.

4 Public construction

The public constructor accepts the same human-readable components as the first design:

```
Time_s( std::uint32_t hours,
        unsigned int  minutes,
        unsigned int  seconds );
```

The arguments specify a non-negative duration. Hours are not limited to 23 because the class represents durations rather than times of day.

DES-3.1 The constructor shall reject a minute argument of 60 or more by throwing `std::invalid_argument`.

DES-3.2 The constructor shall reject a second argument of 60 or more in the same way.

DES-3.3 Once the arguments have been validated, the constructor shall store their combined value in seconds. Conceptually,

```
seconds_ = 3600*hours + 60*minutes + seconds;
```

For example, `Time_s{ 2, 15, 30 }` stores 8130. Unlike the component-based constructor, which can largely copy its arguments into fields, this constructor must perform a conversion.

The conversion is straightforward, but its arithmetic must be performed using a type wide enough to hold the result. Since the public hour argument is a 32-bit unsigned integer, a signed 64-bit integer is sufficient for every valid non-negative constructor input. In the implementation, the hour value is converted to `std::int64_t` before multiplication.

5 Private construction

Arithmetic produces signed totals, including negative values. The class therefore provides a private constructor:

```
Time_s( std::int64_t seconds );
```

DES-4.1 This constructor shall store its argument directly in `seconds_`, without decomposing it into components.

For example, the result of subtracting a duration of two hours from a duration of one hour is the signed integer `-3600`. The private constructor can store that value directly; it does not need to calculate a sign, hours, minutes, and seconds first.

The constructor is private because clients are expected to construct non-negative durations using the public hour-minute-second interface. Negative durations can then be produced using unary minus or arithmetic. The private constructor is an implementation convenience, not an additional public operation.

6 Determining the sign

The public operation

```
bool negative() const;
```

returns whether the duration is negative.

DES-5.1 The implementation shall determine the sign by testing `seconds_ < 0`.

This works even for negative durations shorter than an hour, such as `-0:03:05`, because the sign belongs to the complete stored number rather than to an hour component. For zero, the comparison is false, so no special rule for negative zero is required.

7 Recovering the time components

The public interface exposes the magnitude of the duration as hours, minutes, and seconds:

```
std::uint32_t hours() const;
std::uint8_t  minutes() const;
std::uint8_t  seconds() const;
```

These components are not stored independently. They must be calculated from the magnitude of `seconds_` whenever requested.

If `magnitude` is the non-negative magnitude of the stored value, the components are:

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

For example, the stored value 8130 gives 2 hours, 15 minutes, and 30 seconds. The stored value -8130 must produce exactly the same three magnitude components.

DES-6.1 The accessors shall use the magnitude, not the signed value itself, to calculate the components.

DES-6.2 The hour accessor shall return the whole number of hours; the minute and second accessors shall each return values between 0 and 59.

DES-6.3 None of the accessors shall modify the object.

7.1 Why the magnitude must be calculated first

In C++, division of negative integers truncates toward zero, and a remainder has the sign of its dividend when it is nonzero. For example, `-8130%60` is negative, not positive. Applying the component formulas directly to a negative stored value would therefore produce unsuitable values for unsigned minute and second components.

A further complication arises at the most negative signed 64-bit integer. Its positive magnitude cannot be represented as a `std::int64_t`. The implementation therefore calculates an unsigned magnitude. Conceptually, for a negative value it can use:

```
std::uint64_t magnitude{
    static_cast<std::uint64_t>( -(seconds_ + 1) ) + 1
};
```

Subtracting the extra one *before* negating avoids attempting to negate the smallest signed integer. The unsigned addition then restores the correct magnitude. For non-negative values, conversion to `std::uint64_t` is direct.

This detail makes the component accessors slightly more complicated than those of `Time_hms`, which can simply return stored fields. It also demonstrates that a design simplifying one set of operations can move complexity elsewhere.

The common public interface returns hours as `std::uint32_t`. An arithmetic result could theoretically contain more hours than this return type can represent. The teaching implementation does not check that limit; a comment at the narrowing conversion identifies where a check would be needed in a more general-purpose class. Ordinary human-scale examples remain well within the limit.

8 Comparing durations

Both designs support six comparisons:

```
operator== operator!= operator< operator<= operator> operator>=
```

In the component representation, comparisons must account for the signs and then compare the magnitudes in the correct order. For example, among two negative durations, the one with the larger magnitude is the smaller signed value.

With a single signed count, the ordinary integer comparison already incorporates these rules.

DES-7.1 Each comparison shall compare `seconds_` with `other.seconds_` using the corresponding integer comparison.

For example, `-7200 < -3600` is true, just as `-2:00:00` is less than `-1:00:00`. No separate treatment of positive and negative operands is necessary. Equality is also simpler: two objects represent the same duration precisely when their stored integers are equal.

9 Unary plus and minus

Unary plus returns the current duration unchanged:

```
return *this;
```

Unary minus returns the additive inverse of the duration. The private constructor permits the implementation to express this directly:

```
return Time_s{ -seconds_ };
```

DES-8.1 Unary plus shall return the same value as its operand.

DES-8.2 Unary minus shall return a duration with the opposite sign and the same magnitude; negating zero shall return zero.

The component design must reverse a Boolean sign while ensuring that zero is not made negative. Here, negating the integer zero automatically produces zero.

Negating the smallest `std::int64_t` value would overflow. The simplified implementation does not add a runtime check for this extreme case; a production implementation would need to address it. The examples used in the accompanying demonstration program are nowhere near this limit.

10 Addition and subtraction of durations

The principal advantage of this design appears in arithmetic. Adding two durations means adding their signed total numbers of seconds. Subtracting two durations means subtracting those totals.

DES-9.1 Compound addition shall update the stored value conceptually as follows:

```
seconds_ += other.seconds_;  
return *this;
```

DES-9.2 Compound subtraction shall similarly subtract `other.seconds_` from `seconds_` and return a reference to the modified object.

For example, `1:59:30 + 0:01:30` is computed by adding 7170 and 90, obtaining 7260. The result corresponds to `2:01:00`. There is no need to carry 60 seconds into minutes or 60 minutes into hours; the integer total already describes the correct result.

Similarly, `1:00:00 - 2:00:00` is calculated as `3600 - 7200`, giving `-3600`, or `-1:00:00`. The sign follows automatically from signed subtraction.

10.1 Binary arithmetic reuses compound assignment

The non-modifying binary operators use a copy of the left operand and then apply the corresponding compound-assignment operator:

```
Time_s result{ *this };  
return result += other;
```

Subtraction follows the same pattern with `-=`. This keeps the arithmetic in one place and avoids a recursive dependency between binary and compound-assignment operators.

DES-9.3 Binary addition and subtraction shall leave their operands unchanged and return the computed duration.

DES-9.4 Compound addition and subtraction shall modify the left operand and return a reference to it.

As in the other design, signed arithmetic is assumed to remain representable. The academic implementation does not perform bounds checking on every addition or subtraction. If extreme durations were in scope, checks would need to occur before signed arithmetic that could overflow.

11 Increment and decrement

The public interface includes both prefix and postfix increment and decrement. These operations change a duration by one second, not by one hour or one minute.

DES-10.1 Prefix increment shall increase `seconds_` by one and return a reference to the modified object. Prefix decrement shall decrease it by one and likewise return a reference.

DES-10.2 Postfix increment and decrement shall return the original value while still modifying the object by one second.

For example, if a duration is `0:59:59`, prefix increment changes its stored count from 3599 to 3600, which displays as `1:00:00`. A decrement from zero changes the count from 0 to -1, which displays as `-0:00:01`. No special carrying, borrowing, or sign reversal is required.

The postfix operators first save a copy, apply the corresponding prefix operator, and return the saved copy. This is the same external behavior as in the component-based design, although the underlying update is simpler.

12 String representation

The required display form is:

`[ - ]HHH:MM:SS`

Hours use as many digits as necessary, while minutes and seconds always use two digits. A minus sign appears only for a negative duration.

DES-11.1 The `to_string()` member shall begin with a minus sign precisely when `negative()` is true.

DES-11.2 It shall obtain the magnitude components using `hours()`, `minutes()`, and `seconds()`, formatting minutes and seconds with a leading zero when necessary.

For example, `-305` seconds represents a negative duration of five minutes and five seconds and must display as `-0:05:05`. Zero displays as `0:00:00`, without a minus sign.

The first design has a small advantage here: it already stores the fields needed for formatting. This design must perform division and remainder operations to recover them. The output remains identical, which is the point of the shared requirements specification.

13 Stream output

The non-member insertion operator has the form:

```
std::ostream &operator<<( std::ostream &out,  
                          Time_s const &time );
```

DES-12.1 The operator shall insert the result of `time.to_string()` into the supplied stream and return that stream by reference.

Thus, `std::cout << time` and `time.to_string()` use exactly the same format. No separate formatting algorithm is needed for stream output.

14 Private interface

The private portion of the class is much smaller than that of the component design:

```
private:
    std::int64_t seconds_;

    Time_s( std::int64_t seconds );
```

There is no private sign-and-magnitude constructor, no need for a helper that compares magnitudes, and no need for helpers that add or subtract hours, minutes, and seconds. The one private constructor is enough to create a result directly from a signed count.

The public interface still provides the same kinds of operations as `Time_hms`; it is the implementation behind those operations that differs.

15 Storage considerations

The `Time_s` object contains one 64-bit integer member, which occupies eight bytes. On an ordinary implementation its object size is also eight bytes. Unlike the component-based design, there is no collection of differently sized fields that may require internal padding.

The component design uses a 32-bit hour count, two 8-bit components, and a Boolean sign. Although these fields contain fewer than eight bytes of useful data in total, alignment may increase the size of the complete object. The demonstration program uses `sizeof` to reveal the actual sizes on the system where it is compiled.

The signed 64-bit total has a larger theoretical range of durations than the component design's 32-bit hour field. However, both public interfaces expose hours using `std::uint32_t`. For a fair comparison, the examples remain within that common range; the implementation is not intended to demonstrate extreme numerical limits.

The size of an object is not the only consideration. The component representation may make component access slightly cheaper, while the signed-seconds representation makes arithmetic and comparison much easier to implement. The appropriate representation depends on the operations the class is expected to perform.

16 Design consequences

The first design initially seemed natural because it stored what a human sees: a sign, hours, minutes, and seconds. Its constructor, accessors, and output routine closely followed that representation. But arithmetic required careful carrying, borrowing, magnitude comparisons, and sign handling.

The second design stores a single signed count of seconds. It has several advantages:

- addition and subtraction become ordinary signed integer arithmetic;
- comparisons need no separate sign or magnitude logic;
- positive and negative durations are handled uniformly;
- zero has only one representation;
- increment and decrement require changing one integer by one; and
- the private representation and its invariant are exceptionally simple.

The principal disadvantage is that construction from human-readable components and later retrieval of those components require conversions. The accessors must also calculate an unsigned magnitude carefully when the stored count is negative.

Neither design changes the meaning of the public operations. The demonstration program can apply the same tests to both classes and compare their results. That is the essential lesson of encapsulation: *clients depend on the behavior promised by a class, not on how that behavior is represented internally.*

The broader design lesson is that the representation closest to the way people describe an object is not necessarily the representation that makes its operations simplest. Choosing an internal representation is therefore a decision about which operations should be easy, which invariants must be maintained, and where complexity should reside.