

Date Class

Software design description

Douglas Wilhelm Harder

October 2, 2026

1 Purpose

This document describes the design of the `Date` class specified in the accompanying requirements specification.

The requirements specification describes what the class must do. This document describes how the class will be organized and how the required behavior will be implemented.

The design is intentionally limited to the functionality required by the specification. In particular, this design does not support arithmetic involving an arbitrary number of days or calculating the number of days between two dates.

2 Design Overview

A date will be represented directly by three values:

- a year,
- a month, and
- a day.

These values will be stored as private member variables so that code outside the class cannot arbitrarily modify a valid date.

The class will maintain the following invariant:

Every successfully constructed `Date` object represents a valid date under the calendar rules specified in the requirements.

All operations that modify a date must preserve this invariant.

3 Class Interface

The class will have the following interface:

```
class Date {
public:
    Date( unsigned int year,
          unsigned int month,
          unsigned int day );

    unsigned int year() const;
    unsigned int month() const;
    unsigned int day() const;

    Date &operator++();
    Date  operator++( int );

    Date &operator--();
    Date  operator--( int );

    std::string to_string() const;

private:
    unsigned int year_;
    unsigned int month_;
    unsigned int day_;

    static bool is_leap_year( unsigned int year );

    static unsigned int days_in_month(
        unsigned int year,
        unsigned int month
    );

    static bool is_valid( unsigned int year,
                          unsigned int month,
                          unsigned int day );
};

std::ostream &operator<<( std::ostream &out,
                          Date const &date );
```

The public portion of the class implements the interface specified by the requirements. The private portion contains the representation and helper functions required by this design.

4 Internal Representation

DES-1.1 The year shall be stored in the private member variable

```
unsigned int year_;
```

DES-1.2 The month shall be stored in the private member variable

```
unsigned int month_;
```

DES-1.3 The day shall be stored in the private member variable

```
unsigned int day_;
```

DES-1.4 No public operation shall provide direct write access to these member variables.

5 Class Invariant

DES-2.1 For every successfully constructed `Date` object,

```
year_ >= 1
```

shall be true.

DES-2.2 The value of `month\_` shall be between 1 and 12, inclusive.

DES-2.3 The value of `day\_` shall correspond to a valid day within the specified month and year.

DES-2.4 No `Date` object shall represent a date from October 5 through October 14, 1582.

DES-2.5 Every public operation shall preserve the class invariant.

6 Construction

The constructor will initialize the three member variables from its arguments.

DES-3.1 The constructor shall verify that the supplied year, month, and day represent a valid date.

DES-3.2 Validation shall be performed using the private `is\_valid()` helper function.

DES-3.3 An assertion shall be used to enforce the precondition that the arguments describe a valid date.

Conceptually, construction will perform the following operations:

```
Date::Date( unsigned int year,
            unsigned int month,
            unsigned int day ):
year_{ year },
month_{ month },
day_{ day } {
    assert( is_valid( year_, month_, day_ ) );
}
```

7 Leap Years

Determining whether February contains 28 or 29 days requires knowing whether the specified year is a leap year.

This calculation is an implementation detail and will therefore be provided by a private static member function:

```
static bool is_leap_year( unsigned int year );
```

The function is static because determining whether a year is a leap year does not require a particular `Date` object.

DES-4.1 For years before 1582, the Julian-calendar rule shall be used: a year is a leap year if it is divisible by 4.

DES-4.2 For years after 1582, the Gregorian-calendar rule shall be used: a year is a leap year if

- it is divisible by 400, or
- it is divisible by 4 but not by 100.

DES-4.3 The leap-year calculation shall be used when determining the number of days in February.

8 Days in a Month

The number of valid days in a month will be determined by the private static member function

```
static unsigned int days_in_month(  
    unsigned int year,  
    unsigned int month  
);
```

DES-5.1 The function shall return 31 for January, March, May, July, August, October, and December.

DES-5.2 The function shall return 30 for April, June, September, and November.

DES-5.3 For February, the function shall return 29 during a leap year and 28 otherwise.

DES-5.4 The month supplied to this function shall be required to be between 1 and 12, inclusive.

9 Date Validation

The private static member function

```
static bool is_valid( unsigned int year,  
                     unsigned int month,  
                     unsigned int day );
```

will determine whether three values describe a valid date.

DES-6.1 The function shall return **false** if the year is zero.

DES-6.2 The function shall return **false** if the month is outside the range 1 through 12.

DES-6.3 The function shall return **false** if the day is zero or is greater than the number of days in the specified month.

DES-6.4 The function shall return **false** for dates from October 5 through October 14, 1582.

DES-6.5 The function shall return **true** otherwise.

10 Date Components

The public member functions

```
unsigned int year() const;
unsigned int month() const;
unsigned int day() const;
```

will provide read-only access to the corresponding private member variables.

DES-7.1 Each function shall directly return the corresponding member variable.

DES-7.2 These functions shall be declared `const` because they do not modify the object.

11 Incrementing a Date

Prefix increment shall be implemented by

```
Date &operator++();
```

and shall move the object to the immediately following valid calendar date.

DES-8.1 If the current date is October 4, 1582, incrementing the date shall change it directly to October 15, 1582.

DES-8.2 Otherwise, if another day exists within the current month, the day shall be incremented.

DES-8.3 If the current date is the final day of a month other than December, the day shall become 1 and the month shall be incremented.

DES-8.4 If the current date is December 31, the day and month shall become 1 and the year shall be incremented.

Thus, the general logic is

```
if the date is October 4, 1582
    move to October 15, 1582
else if another day exists in the current month
    increment the day
else if the current month is not December
    set the day to 1
    increment the month
else
    set the day to 1
    set the month to January
    increment the year
```

DES-8.5 Prefix increment shall return a reference to the modified object.

12 Postfix Increment

Postfix increment shall be implemented by

```
Date operator++( int );
```

DES-9.1 Postfix increment shall first make a copy of the current object.

DES-9.2 The current object shall then be incremented using prefix increment.

DES-9.3 The saved copy shall be returned.

Conceptually,

```
Date Date::operator++( int ) {  
    Date old_date{ *this };  
    ++(*this);  
    return old_date;  
}
```

This avoids duplicating the date-incrementing logic.

13 Decrementing a Date

Prefix decrement shall be implemented by

```
Date &operator--();
```

and shall move the object to the immediately preceding valid calendar date.

DES-10.1 If the current date is October 15, 1582, decrementing the date shall change it directly to October 4, 1582.

DES-10.2 Otherwise, if the day is greater than 1, the day shall be decremented.

DES-10.3 If the date is the first day of a month other than January, the month shall be decremented and the day shall become the final valid day of the new month.

DES-10.4 If the date is January 1 of a year greater than 1, the year shall be decremented, the month shall become December, and the day shall become 31.

DES-10.5 The date January 1, 1 has no preceding date within the supported domain.

Attempting to decrement this date shall violate a precondition and shall be detected using an assertion.

DES-10.6 Prefix decrement shall return a reference to the modified object.

14 Postfix Decrement

Postfix decrement shall be implemented by

```
Date operator--( int );
```

DES-11.1 Postfix decrement shall first make a copy of the current object.

DES-11.2 The current object shall then be decremented using prefix decrement.

DES-11.3 The saved copy shall be returned.

Conceptually,

```
Date Date::operator--( int ) {  
    Date old_date{ *this };  
    --(*this);  
    return old_date;  
}
```

15 String Representation

The member function

```
std::string to_string() const;
```

will produce the required year-month-day representation.

DES-12.1 The month shall always occupy two decimal digits.

DES-12.2 The day shall always occupy two decimal digits.

DES-12.3 The components shall be separated by hyphens.

For example,

```
2026-10-02
```

shall be produced for October 2, 2026.

DES-12.4 The function shall not modify the object.

16 Stream Output

The stream insertion operator shall be implemented as a free function:

```
std::ostream &operator<<( std::ostream &out,  
                          Date const &date );
```

It cannot be a member of `Date` because its left operand is an output stream.

DES-13.1 The operator shall insert the string returned by

```
date.to_string()
```

into the supplied output stream.

DES-13.2 The operator shall return the output stream by reference so that stream insertion operations may be chained.

Conceptually,

```
std::ostream &operator<<( std::ostream &out,  
                          Date const &date ) {  
    out << date.to_string();  
    return out;  
}
```

17 Traceability

The following table summarizes how the major requirements are addressed by this design.

Requirement	Design element
Construction	Constructor and <code>is_valid()</code>
Valid calendar dates	<code>is_valid()</code> , <code>is_leap_year()</code> , and <code>days_in_month()</code>
Date components	<code>year()</code> , <code>month()</code> , and <code>day()</code>
Next date	Prefix and postfix <code>operator++</code>
Previous date	Prefix and postfix <code>operator--</code>
String representation	<code>to_string()</code>
Stream output	<code>operator<<</code>

18 Design Limitations

This design deliberately stores the year, month, and day directly.

This representation makes the operations required by the current specification straightforward and keeps the relationship between the calendar date and its internal representation easy to understand.

The design is not intended to efficiently support operations such as

```
date + 100000  
date - 100000  
date1 - date2
```

A subsequent design may use a different internal representation or additional algorithms to support such operations efficiently.