Date class

You will now implement a date class. First, you will be asked to design one. We will then provide a design, and you are welcome to either continue implementing yours, you can update yours to use ideas from ours, or you can proceed with ours. You will then test your implementation.

Class declaration

The class declaration is straightforward:

```
class Date;
```

Class definition: member variables

The come up with a class definition that allows us to store a date; that is, a year, a month and a day. You can describe your representation. Also, we would like to describe how you would define an *invalid* date; that is, a date object that is the equivalent of of *not-a-number* to represent an invalidly defined date. To be fair, you are welcome to implement any calendar you wish; however, we will be implementing the Gregorian calendar.

```
class Date {
    private:
        // List and describe your member variables here
};
```

An example declaration of member variables

Below is the model we will consider.

```
class Date {
    private:
        // The year is any non-zero integer with
        //     1 representing 1 CE and
        //    -1 representing 1 BCE
        int            year_;
        // A value between 1 and 12 representing
        // January through December, respectively.
        short month_;
        // A value between 1 and 28, 29, 30 or 31,
        // depending on the mont hand year.
        short day_;
};
```

Any Date with `year_ == 0` is an invalid date. For any date where the year is non-zero, the date must be possible, so a date 2021-2-29 would be invalid, as 2021 was not a leap year, and thus there is no February 29th that year.

An alternative implementation would be to simply store a signed long integer, and this would represent the number of days from January 1st, 1 CE including that date, where -1 would represent December 31st, 1 BCE. An invalid date would be when this value was exactly equal to zero.

Class definition: member functions and operators

Next, come up with appropriate member functions to make this date class useful. Below you can extend our implementation, but you can also choose to extend yours, instead.

```
class Date {
    public:
        // List and describe your member functions here

    private:
        // The year is any non-zero integer with
        //     1 representing 1 CE and
        //    -1 representing 1 BCE
        int   year_;
        // A value between 1 and 12 representing
        // January through December, respectively.
        short month_;
        // A value between 1 and 28, 29, 30 or 31,
        // depending on the mont hand year.
        short day_;
};
```

An example declaration of member functions and operators

Below is the model we will consider.

```
class Date {
    public:
        Date();      // Creates an invalid date 0-0-0
        Date(
            int            new_year,
            short new_month,
            short new_day
        );

        // No destructor required
        //  - All memory is automatically deallocated
        // Default copy constructor
        //  - Just copy over all three member variables
        //  - No move constructor required
        // Default assignment operator
        //  - Just copy over all three member variables
        //  - No move operator required

        // Retrieve the year – 0 for an invalid date
        int year() const;
        // Retrieve the month - 0 for an invalid date
        short month() const;
        // Retrieve the day – 0 for an invalid date
        short day() const;

        // Set the date, but if the date is
        // invalid, set all three to 0
        void date(
            int            new_year,
            short new_month,
            short day
       );

       // Return "January" through "December"
       std::string month_to_string() const;

       // Is the current year a leap year?
       bool is_leap_year() const;

       // How many days in the current month
       short days_in_month() const;

       // Is this date a valid one?
       bool is_valid() const;
```

```
// This just returns the date
Date operator+() const;

// Return a date equal to this date
// plus the given number of days
Date operator+( int days ) const;

// Return a date equal to this date
// minus the given number of days
Date operator-( int days ) const;

// Calculate the number of days between
// this date and the right-hand side date
int operator-( Date const &rhs ) const;
// Decrease or increase the number
// of months by the given number
Date operator<<( unsigned int n ) const;
Date operator>>( unsigned int n ) const;

// Are the dates equal or unequal?
bool operator==( Date const &rhs ) const;
bool operator!=( Date const &rhs ) const;

// Is this date before, before or equal to,
// after or equal to, or after the right-hand
// date?
bool operator< ( Date const &rhs ) const;
bool operator<=( Date const &rhs ) const;
bool operator>=( Date const &rhs ) const;
bool operator> ( Date const &rhs ) const;

// Increment or decrement this date to
// the next day, returning a reference to this
Date &operator++();
Date &operator--();
// Increment or decrement this date to
// the next day, but return the original
// date prior to the increment or decrement
Date operator++( int );
Date operator--( int );

// Same as the above
Date &operator+=( int n );
Date &operator-=( int n );
Date operator>>=( unsigned int n );
Date operator<<=( unsigned int n );
```

```
        private:
            int   year_;
            short month_;
            short day_;
    };
```

The member operators

```
               Date operator+( int days ) const;
               Date operator-( int days ) const;
```

are only called if you have the integer on the right-hand side, so

```
    Date some_day{ 5, 3, 1969 };
    Date another_day{ some_day + 20 };
```

This member operator is not called if you calculate:

```
    Date another_day{ 20 + some_day };
```

To do this, you need to declare a non-member function

```
    Date operator+( int days, Date const &date ) {
        return date + days;
    }
```

Note what happens: because the addition of an integer to a date is commutative, that is,

$$n + day = day + n,$$

we just tell the compiler to call the corresponding member operator defined above.

There will also be one function that is declared:

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

This should treat the out parameter as if it was `std::cout`, so you wanted to print a date using a function; for example,

```
void print_date( Date const &date ) {
    std::cout << date.day() << " " << date.month_to_string()
              << ", " << date.year();
}
```

Instead, you would do exactly the same in this operator, only you should return the out object:

```
std::ostream &operator<<(
  std::ostream &out,
  Date const   &date
) {
    out << date.day() << " " << date.month_to_string()
        << ", " << date.year();

    return out;
}
```

In the first example, you might call:

```
Date some_date{ 5, 3, 1969 };
print_date( some_date );
std::cout << std::endl;
```

now, you can call:

```
Date some_date{ 5, 3, 1969 };
std::cout << some_date << std::endl;
```

What's nice, but beyond the scope of this course, is there are other objects that print to files and not the console, and if you call that alternative object (also an object of the class `std::ostream`), it will work in exactly the same manner.

You will note that while we do allow users to subtract an integer from a date, we do not allow for a date to be negated or for a date to be subtracted from an integer. This is because it makes no sense to negate a date: what is the negation of March 5th, 1969? Would it be March 5th, 1969 BCE, or would it be as many days BCE as March 5th, 1969 CE is days into the common era? However, as January 1st, 1 CE is an arbitrarily chosen date, linked to the New Year of the older Julian calendar and an inaccurate selection of the birth of one individual, neither of these make any sense, so while we can calculate the difference between two dates, we cannot negate a year. Thus, in arithmetic, we have

$$a + b = c \text{ if and only if } c - a = b \text{ if and only if } -a - b = -c$$

$$\text{if and only if } c - b = a \text{ if and only if } a - c = -b$$

however, with dates and integers, we are restricted to:

$$date1 - date2 = n \text{ if and only if } \text{date1} - n = date2 \text{ if and only if } date1 = \text{date2} + n$$

Note that an operation such as the addition of two dates make as much sense as the addition of two addresses (pointers). You can calculate the difference between two addresses, but it makes no sense to add two addresses. The use of >> and << to change a date by the month allows the user to not have to calculate how many days in the next month. In this same manner, one can increment or decrement a year by right- or left-shifting by a multiple of twelve months.

It is now up to you to both implement and test your implementation.