Module IN3013/INM173 – Object-Oriented Programming in C++

Solutions to Exercise Sheet 3

1. Output of `Date`s is very similar to output of `Point`s, as discussed in the lecture.

```
ostream & operator<<(ostream & out, const Date & d) {
        out << d.day() << '/' << d.month() << '/' << d.year();
        return out;
}
```

   As in the case of `Point`s, we have no choice but to make this an independent function. We cannot add it as a method to the `ostream` class, because that was defined in the standard library.

2. We can use exactly the same `<<` operator for the new representation of `Date`s, because it uses only public methods of the class, and that has not changed.

   Note that if we had used public fields instead of methods, changing the representation would have forced us to change any client code, like the `<<` operator.

3. A `-` operator can be implemented either

   - as a member function of `Date`, because its first argument is a `Date`, or
   - as an independent function, because it can be defined using only public member functions (in this case `day_number`).

   So we add to the `Date` class the definition

```
long operator-(Date d) const {
        return julian_day - d.day_number();
}
```

   The method is `const` because it does not change the object it is invoked on.

   This method is simple enough to be defined in place in the class. (For people who worry about efficiency: this method will then be inlined, and so will `day_number`, so it will be compiled to a simple subtraction.)

   (a) The number of days to Christmas:

```
Date today;
Date christmas(25, 12);
cout << "Only " << (christmas - today) << " days to Christmas\n";
```

The variable `today` uses the default constructor, which initializes it to today's date. The variable `christmas` is initialized to the 25th of December in the current year. The difference is the number of days till Christmas.

We could have used temporary objects instead of defining object variables:

```
cout << "Only " << (Date(25, 12) - Date()) << " days to Christmas\n";
```

(b) My son's age in days:

```
Date birthday(21, 12, 1995);
cout << "Jason is " << (today - birthday) << " days old.\n";
```

Note that this datatype expects years to be specified in full (95 means 95 AD). The variable `today` was defined above.

4. Adding a `Date` to an `int` can be defined either as a member function of `Date`:

```
Date operator+(int n) const {
        return Date(julian_day + n);
}
```

or as an independent function:

```
Date operator+(Date d, int n) {
        return Date(d.day_number() + n);
}
```

However adding an `int` to a `Date` can only be defined as an independent function, because `int` is not a class:

```
Date operator+(int n, Date d) {
        return Date(n + d.day_number());
}
```

(a) Tomorrow's date:

```
cout << "Tomorrow is " << (today+1) << '\n';
```

(b) The date 28 days from today id `today + 28`. Because dates are internally represented as day numbers, changing months is no problem.

(c) The day when you'll be twice as old (in days) as you are today:

```
cout << (birthday + 2*(today - birthday)) << '\n';
```

(d) The day before 1 Jan 1 is 31 Dec -1 (i.e. 1 BC), because 1 AD immediately followed 1 BC – there was no year 0.

(e) The day after 2 Sep 1752 was 14 Sep 1752, because 11 days of were dropped from the calendar on the switch from the Julian calendar to the Gregorian one.

Often one defines a `+=` operator, because using it avoids creating temporary objects (not an issue here, because `Date` objects are small). The `+=` operator for `Date`s is the member function

```
void operator+=(int n) {
        julian_day += n;
}
```

Such operators are usually written as member functions.

This is good enough to use the assignment as a statement, but not to use it inside an expression. Doing it properly is a bit more complicated; we'll do overloaded assignment in sessions 8 and 9.

The `-=` operator is similar.

5. The comparison operators for for `Date`s may be defined as member functions of the class `Date`:

```
bool operator==(Date d) const {
        return julian_day == d.day_number();
}

bool operator<(Date d) const {
        return julian_day < d.day_number();
}
```

or as independent functions:

```
bool operator==(Date d1, Date d2) const {
        return d1.day_number() == d2.day_number();
}

bool operator<(Date d1, Date d2) const {
        return d1.day_number() < d2.day_number();
}
```

The dates for today, tomorrow, etc up to the end of the year:

```
Date end_of_year = Date(31, 12);
for (Date d = today; d <= end_of_year; d = d + 1)
    cout << d << '\n';
```

If we have a `+=` operator, we can write

```
Date end_of_year = Date(31, 12);
for (Date d = today; d <= end_of_year; d += 1)
    cout << d << '\n';
```

We cannot write `++d`, unless we give a definition of `++` for dates.

6. Like the output operator, the input operator `>>` must be written as an independent function:

```
istream & operator>>(istream & in, Date & date) {
        int d, m, y;
        char c1, c2;
        if (in >> d)
                if (in >> c1 >> m >> c2 >> y &&
                    c1 == '/' && c2 == '/')
                        date = Date(d, m, y);
                else
                        in.set(ios::badbit); // corrupted
        return in;
}
```

We expect to read the date in the same form are the `>>` operator writes it. First we read the day number `d`. If there is any failure after that (either early end of input or unexpected input) then the stream will have been corrupted: we will have read data that we have not stored in `date`. Then we expect a character (which should be a slash), a month number, another character (which should also be a slash), and a year number. Only if all of that succeeds to we set the date, otherwise the stream is corrupted.

In any case, we return the stream `in`; the caller should test it to see whether data was read.

7. The alternative representation would have private state

```
long julian_day;
int d, m, y;
```

The constructors would be changed to initialize all of these

```
Date() : julian_day(julian_today()) {
        julian_to_date(julian_day, d, m, y);
}

Date(int dd, int mm) : d(dd), m(mm), y(Date().year()) {
        julian_day = date_to_julian(d, m, y);
}

Date(int dd, int mm, int yy) : d(dd), m(mm), y(yy) {
        julian_day = date_to_julian(d, m, y);
}

Date(long jd) : julian_day(jd) {
        julian_to_date(jd, d, m, y);
}
```

The extractor functions would be defined as in the initial version:

```
// The day of the month (1-31)
int day() const { return d; }

// The month of the year (1-12)
int month() const { return m; }

// The year number
int year() const { return y; }
```

The only other change would be to methods that changed the state, like the `+=` operator, which would have to update the rest of the state correspondingly:

```
void operator+=(int n) {
        julian_day += n;
        julian_to_date(julian_day, d, m, y);
}
```