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

Solutions to Exercise Sheet 9

1. I will show how to split up the file `dates.cc` into a header file and a C++ source file. The idea is that the source file will be separately compiled and linked with other parts of the program. The header file will be included by any other parts of the program that use the `Date` class, and they must be recompiled whenever the header file changes, so we want to make it as small as possible. Another reason for making it small is an engineering one: the header file is the external interface of the `Date` class that is known by clients. By limiting that, we have greater flexibility in changing the implementation of the `Date` class without affecting clients.

   I am following the common convention of one header and source file for each class (here `Date`), both named after the class. This is similar to the Java convention, and certainly makes the organization easy to follow. However, it is not required by C++, and sometimes there are good reasons to depart from the convention. If some class is used only in the implementation of another, it could be placed in the same file. If classes are very closely coupled, they could be put in the same file.

   Note: In the last coursework, I asked you to include `dates.cc` as a quick and dirty way of putting two source files together to make a program. The *right* way is to use separate compilation. Including a `.cc` file is bad style, and almost always an error.

   Date.h There are many ways to split up `dates.cc` into a header and source file. I have chosen a fairly extreme approach, with little more that interface in the header file. In particular, I consider the various Julian conversion functions to be part of the implementation, and hide them in `Date.cc`. Those functions are used in some of the constructors, so the code for those constructors (including initializers) must also be split off from the class and moved into `Date.cc`. (An alternative is to declare the Julian functions as `extern` and define them in `Date.cc`, but I've chosen to hide even their names. I have however left the code of one constructor and two simple methods in the class. I could have moved even these out into `Date.cc`.

```
#ifndef DATE_H
#define DATE_H

#include <iostream>

using namespace std;
```

```
class Date {
        // The date as a Julian day number
        long julian_day;

public:
        // Today's date
        Date();

        // A date in the current year
        Date(int d, int m);

        // A fully specified date
        Date(int d, int m, int y);

        // The date of a Julian day number
        Date(long jd) : julian_day(jd) {}

        // The day of the month (1-31)
        int day() const;

        // The month of the year (1-12)
        int month() const;

        // The year number
        int year() const;

        // The day of the week (Sun=0, ..., Sat=6)
        int day_of_week() const { return (julian_day + 1)%7; }

        // The date as a Julian day number
        long day_number() const { return julian_day; }

        Date operator+(int n) const;
        Date operator-(int n) const;
        int operator-(Date d) const;

        Date &operator+=(int n);
        Date &operator-=(int n);

        bool operator==(Date d) const;
        bool operator<(Date d) const;
};
```

```
extern Date operator+(int n, Date d);

extern ostream & operator<<(ostream & out, const Date & d);
extern istream & operator>>(istream & in, Date & date);

#endif
```

I also have to declare the independent functions (all operators here) that clients will use. The syntax for that is to prefix the declaration with `extern` and replace the body by a semicolon. Clients see the type of the operators, which is all they need.

The header file needs to include `<iostream>` because it uses the names `istream` and `ostream`. It should not rely on the including file having included `<iostream>` before including `Date.h`.

Finally, the whole file is wrapped in an include guard. These should always be the first two lines and the last line of each header file. It is then same to include the header file several times: only the first will count.

Date.cc The corresponding C++ source file contains the implementations of the things declared in the header file, plus any auxilliary stuff used by these implementations. Here is will contain the implementations of methods, and of the independent operators, as well as other functions and constants they use. The file begins with inclusions of system headers and user headers:

```
#include <cassert>
#include <cmath>
#include <ctime>
#include <iostream>
#include "Date.h"

using namespace std;
```

The inclusion of `<iostream>` is optional, because `Date.h` already includes it, but perhaps it is a little clearer. It does no harm (because the header file has an include guard).

Next we have the various auxilliary constants and functions that the methods and operators use.

```
// Julian day numbers, representing the number of days since 1 Jan 4713BC,
// are used to avoid the various quirks of calendars, and to simplify
// various calculations with dates.
```

```
// The time function returns the number of seconds from the start of
// 1 Jan 1970 to now, from which we can compute today's Julian day number.

const long seconds_per_day = 24*60*60L;

// Day number when time() == 0 at start of day (1 Jan 1970)
const long time_epoch = 2440588L;

long julian_today() {
        return time(0) / seconds_per_day + time_epoch;
}

// Up to 2 Sep 1752, Britain and colonies used the Julian calendar.
// From the next day, they used the Gregorian calendar, in which that
// day was 14 Sep 1752 (i.e. 11 days were dropped).

// Day number of first day of Gregorian calendar in Britain (14 Sep 1752)
const long gregorian_epoch = 2361222L;

// Convert a date to a Julian day number
long date_to_julian(int d, int m, int y) {
        // there was no year 0
        if (y < 0)
                y++;
        // consider Jan and Feb as belonging to previous year
        if (m <= 2) {
                y--;
                m += 12;
        }
        m++;
        long start_of_year = 1720995L + (long)floor(365.25 * y);
        int day_of_year = (long)floor(30.6001 * m) + d;
        long jd = start_of_year + day_of_year;
        // Gregorian correction
        if (jd >= gregorian_epoch)
                jd += 2 + y/400 - y/100;
        return jd;
}

void julian_to_date(const long jd, int &d, int &m, int &y) {
        if (jd >= gregorian_epoch)
                // JD 1721120 = 1 Mar 1BC (Gregorian calendar)
```

```
                y = (int)floor((jd - 1721120) / 365.2425);
        else
                // JD 1721118 = 1 Mar 1BC (Julian calendar)
                y = (int)floor((jd - 1721118) / 365.25);
        long start_of_year = 1720995L + (long)floor(365.25 * y);
        // Gregorian correction
        if (jd >= gregorian_epoch)
                start_of_year += 2 + y/400 - y/100;
        int day_of_year = jd - start_of_year;
        m = (int)floor(day_of_year / 30.6001);
        d = day_of_year - (long)floor(30.6001 * m);
        // consider Jan and Feb as belonging to previous year
        m--;
        if (m > 12)
                m -= 12;
        if (m <= 2)
                y++;
        // there was no year 0
        if (y <= 0)
                y--;
        assert(date_to_julian(d, m, y) == jd);
}
```

Now we are ready to define the methods declared in the header file. First, the three constructors we extracted from the header file:

```
Date::Date() : julian_day(julian_today()) {}
Date::Date(int d, int m) :
        julian_day(date_to_julian(d, m, Date().year())) {}
Date::Date(int d, int m, int y) :
        julian_day(date_to_julian(d, m, y)) {}
```

Then some extractor methods:

```
int Date::day() const {
        int d, m, y;
        julian_to_date(julian_day, d, m, y);
        return d;
}

int Date::month() const {
        int d, m, y;
        julian_to_date(julian_day, d, m, y);
```

```
        return m;
}

int Date::year() const {
        int d, m, y;
        julian_to_date(julian_day, d, m, y);
        return y;
}
```

Next we come to the operators defined as methods of the class (because their first argument was a `Date`):

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

Date Date::operator-(int n) const {
        return Date(julian_day - n);
}

int Date::operator-(Date d) const {
        return (int)(julian_day - d.day_number());
}

Date & Date::operator+=(int n) {
        julian_day += n;
        return *this;
}

Date & Date::operator-=(int n) {
        julian_day -= n;
        return *this;
}

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

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

Some other operators had to be defined as external functions (because their first argument was not a `Date`). These were declared with `extern` in the header file.

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

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

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

Other files The general rule for deciding whether or not to include is: if a file mentions a name, it should include the header file in which that name is defined. For example, the declaration of the appointment class (in `Appointment.h`) will probably mention `Date`, so `Appointment.h` should include `Date.h`. If some other class uses `Date` only in its implementation, then only its `.cc` file would include `Date.h`.

There would usually be another source file containing the `main` function. Since this is not called from elsewhere in the program, it will not need a header file (unless it provides other things that are used elsewhere).

2.

3.

4. If you change a source file, that file gets recompiled, and then all the object files are relinked to create the executable.

   If you change a header file, all the source files that include it (directly or indirectly) get recompiled, and the program is relinked.

5. It is the usual problem with redundancy: if the same information is given in two places, one must be very careful to keep them the same. In the case of a function declaration, if one is changed, you may have a program that is accepted by the

compiler, but is calling the function with the wrong number of arguments, or with arguments of the wrong type, which may lead to data corruption and possibly a program crash.