IB9N7 C++ for Quantitative Finance
Worksheet 8

Introduction to Object Oriented Programming
Hints and solutions

30 November 2015
(Week 9)

Keep records of your solutions and show them to us when you have finished.
If you do not understand (or you disagree with) the hints and solutions, please ask us at the start of the next lab session. Remember that you are not necessarily expected to be able to solve everything without help.
To help us judge the timing, please let us know how long it takes you to do the exercises. If you are unable to finish the exercises during the lab session, please finish them before next time.

Objectives for this lab session

By the end of this session, you should have completed the following:

- Understand the program that demonstrates the `Date` class!
- Add/think about appropriate additional functionality for that class.
- Create a `Point` class that has appropriate member variables and functions with the appropriate access restrictions and validation.

Exercises

Exercise 1: Understanding

(a) Download, extract, compile and run the (complete version of the) `Date` example from the lecture `08_date.zip`. (It would be confusing for dev-cpp to try to open the files without extracting them.)

(b) Try to understand it. Focus on one member function at a time. Play with the debugger if you like.

(c) (i) What would happen if you tried to access the private members of the `Date` class from the `date_test_main` function?
e.g. `test.day_`

```
In file included from Date_main.cpp
'void date_test_main()':
[Error] 'int Date::day_' is private
[Error] within this context
```

- The first line of the error message shows you the location of the `#include` directive.

- The second line takes you to the header file.
- The third line takes you to the definition of day_.
- The last line takes you to the context in which you attempted to use day_, and in this instance, this is the line you would need to fix (by replacing day_ with get_day()).

(ii) What would happen if you tried to access the non-static members of the Date class from a static context?

e.g. Date::output_shortform() from date_test_main or output_shortform() from is_leap_year.

There would be no Date object instance on which to act, i.e. there would be no day_, month_ or year_ to output.

Even if the non-static function output_shortform() didn't attempt to access day_, month_ or year_ etc, the compiler would still reject attempts to invoke a non-static function from a static context.

```
In function 'void date_test_main()':
[Error] cannot call member function 'void Date::output_shortform() const'
    without object
```

(iii) What would happen if you tried to access the static members of the Date class from a non-static context?

e.g. test.is_valid_date(1, 1, 2014) from date_test_main

This is valid C++ code, but the object instance that is referred to is (mostly) irrelevant beyond knowing its static type.

It is advisable never to call static members using the component selection operator on an object instance, but instead to use the scope resolution operator on the class name.

(d) (i) Write a free function that takes a Date as an argument and returns a Date corresponding to the following day.

```cpp
// This is Date_next.cpp.
// Alternatively, could include this as part of Date.cpp or Date_main.cpp
#include "Date.hpp"
#include "Date_next.hpp"

Date next_date(Date start)
{
    int day = start.get_day();
    int month = start.get_month();
    int year = start.get_year();

    // best to avoid having any month == 13 etc at any instant
    // but not a disaster in a free function

    // If it's not the last day in the month...
    if (day != Date::month_length(month, year))
    {
        // ... calculating the next day is simple.
        ++day;
    }
    else
    {
        // Otherwise, we want the first day of the following month.
        day = 1;
        // Which is also easy unless it takes us into the following year...
        if (month != 12)
        {
            ++month;
        }
        else
        {
            month = 1;
            ++year;
        }
    }

    return Date(day, month, year);
}
```

(ii) Write a member function that does the same thing.
This should not modify the current Date.

```
// This should be part of Date.cpp with appropriate changes to Date.hpp made as well.
Date Date::next_date() const
{
    int day = get_day();
    int month = get_month();
    int year = get_year();

    if (day != month_length(month, year))
    {
        ++day;
    }
    else
    {
        day = 1;
        if (month != 12)
        {
            ++month;
        }
        else
        {
            month = 1;
            ++year;
        }
    }

    return Date(day, month, year);
}
```

As discussed in lecture 9, it may be preferable to provide this as a free function instead of as a member function.

(iii) Now write a member function that advances the Date of the current object instance by one day.

```
// This should be part of Date.cpp with appropriate changes to Date.hpp made as well.
void Date::advance_date()
{
    if (day_ != month_length(month_, year_))
    {
        ++day_;
    }
    else
    {
        day_ = 1;
        if (month_ != 12)
        {
            ++month_;
        }
        else
        {
            month_ = 1;
            ++year_;
        }
    }
}
```

Again, as discussed in lecture 9, it may be preferable to provide this as a free function instead of as a member function.

(iv) Think about what this kind of function (one returning the day following a given date) might look like if you could not use object-orientation at all.

(e) The `Date` class is currently very basic and there are lots of extra things that it could (and in some cases, should) be extended to do.

Think of some extra functionality that you might want the Date class to provide, and what additional members or non-members you might need to provide that functionality.

If you feel that you have the time and the necessary knowledge, try to provide some (but not necessarily all) of that functionality.

There are lots of things. For example:

- compare if two dates are equal
- determine if one date occurs before/after another
- calculate the difference between two dates
- add/subtract more than one day to a date (e.g. *n* days or *n* months or *n* years)
- finding the number of days in a given year

Possible implementations for some of these, with thanks to Jeremy:

```
//additions to the class definition
//class Date
//{
//      public:
//   // ... as before ...
          //static int year_length(int year);
          //void add_days(int days);
          //void add_months(int months); //or, if new date invalid, closest date before when months > 0, and
     closest date after when months < 0
          //void add_years(int years); // similarly
          //void get_all(int& day, int& month, int& year);
          //bool is_equal(const Date& o) const
          //bool is_before(const Date& other) const;
          //int get_days_until(const Date& other) const;
//};

int Date::year_length(int year)
{
    return is_leap_year(year) ? 366 : 365;
}

void Date::get_all(int& day, int& month, int& year) const
{
    day = day_;
    month = month_;
    year = year_;
}

bool Date::is_equal(const Date& o) const
{
    return year_ == o.year_ && month_ == o.month_ && day_ == o.day_;
}

void Date::add_days(int days)
{
    // Could repeatedly call advance_day but that would be inefficient...
    day_ += days;
    // for positive days added
    while (day_ > month_length(month_, year_))
    {
        // use the number of days in the month (BEFORE incrementing the month)
        day_ -= month_length(month_, year_);
        // increment the month
        if (month_ == 12)
        {
```

```cpp
            month_ = 1;
            ++year_;
        }
        else
        {
            ++month_;
        }
    }
    // for negative days added
    while (day_ < 1)
    {
        if (month_ == 1)
        {
            --year_;
            month_ = 12;
        }
        else
        {
            --month_;
        }
        // use the number of days in the month (AFTER decrementing the month)
        day_ += month_length(month_, year_);
    }
}

void Date::add_months(int months)
{
    int month = month_ + months; // new month, possibly < 0 or > 12
    // Modular arithmetic calculations. Need index to be between 0 and 11 inclusive, instead of 1 to 12.
    int div = (month - 1) / 12; // number of whole years (positive or negative) in the new month value
    int rem = (month - 1) % 12; // number of residual months, though possibly negative
    // If the modulus operator gave us a negative answer, correct it.
    if (rem < 0)
    {
        rem += 12;
        --div;
    }
    // Update state by number of whole years and set to the residual month.
    year_ += div;
    month_ = 1 + rem; // Update from 0 to 11 range back to 1 to 12 range.
    // Check the date is valid. If not, set to last day in the month.
    int length = month_length(month_, year_);
    if (day_ > length)
    {
        day_ = length;
    }
}

void Date::add_years(int years)
{
    year_ += years;
    // Check the date is valid. If not, set to last day in the month.
    int length = month_length(month_, year_);
    if (day_ > length)
    {
        day_ = length;
    }
}

bool Date::is_before(const Date& other) const
{
    // compare the most significant component first
    if (year_ < other.year_)
        return true;
    // so now year_ >= other.year_
    if (year_ > other.year_ || month_ > other.month_)
        return false;
    // so now year_ = other.year_ && month <= other.month_
    if (month_ < other.month_ || day_ < other.day_)
        return true;
    // so now year_ = other.year_ && month == other.month_ && day_ >= other.day_
    return false;
}

int Date::get_days_until(const Date& other) const
{
    // Tricky one...
    // Start with the easiest case where there's nothing to calculate.
    if (is_equal(other))
        return 0;
    // Easier if we can assume this date is before the other date.
    if (!is_before(other))
        return -other.get_days_until(*this);
    Date runner = *this; // create a local copy of the current object instance
    // We will adjust 'runner' until we reach 'other', counting the adjustments that we make.
    // Easier if we don't have to worry about irregular shapes of months
    if (day_ > 28)
    {
        // Do the calculation with runner.day_ == 28
        runner.add_days(28 - day_);
        // and adjust the result as necessary.
        return runner.get_days_until(other) + 28 - day_;
    }
    int daysMoved = 0;
    // Keep running until the two dates are in the same year
    while (runner.year_ < other.year_)
    {
        if (runner.month_ == 1 || runner.month_ == 1)
            daysMoved += year_length(runner.year_);
        else
            daysMoved += year_length(runner.year_+1);
        runner.add_years(1);
    }
    // It may no longer be the case that runner.is_before(other)
    if (other.is_before(runner))
        return daysMoved - other.get_days_until(runner);
    // Keep running until the two dates are in the same month (already in the same year)
    while (runner.month_ < other.month_)
    {
        daysMoved += month_length(runner.month_, runner.year_);
        runner.add_months(1);
    }
    // The final calculation is correct regardless of whether runner.is_before(other)
    return daysMoved + other.day_ - runner.day_;
}
```

(These have not been fully tested. It is more likely than usual that there may be some mistakes! Let us know if you find any.)

Exercise 2: Creating a new class

(This exercise may take some time but will help you to understand classes more. However, the class test will not involve you having to create your own classes — but the project might!)

(a) Design a class called `Point` to represent a point in a 2-dimensional Euclidean space. Do not start to implement any of the members until you have finished specifying all the entities to be part of the class.

You should think about:

- What data it needs.
- What accessors and mutators to provide.
- What validation needs to be performed.

- What methods it might be useful to have.
- Whether there is any need for any static methods.

Hints:

- A point can be specified in a number of co-ordinate systems. For example, the Cartesian co-ordinates $(x = 0, y = 1)$ and the polar co-ordinates $(r = 1, \theta = \pi/2)$ are two ways of representing the same point.
- C++ has a function `std::atan2(y, x)` for the principal value of the arc tangent of `y/x`, expressed in radians.
- Are there any co-ordinates in either representation that should be rejected (e.g. they represent invalid points)? No points are invalid, with the possible exceptions of points with one or more components that have value ∞ or NaN. However, the `Point` class may be more useful if it actually allows such points rather than rejecting them, in the same way that the floating point types normally allow such values and handle them sensibly.
- Are representations in each form unique? If not, what are the implications for your class? Not unique, consider range of the angle. `p.set_theta(t); new_t = p.get_theta();` may result in `t` and `new_t` not being the same.
- One useful method might be to calculate the distance between the point and some other point.

(b) Implement and test your class!

See `08_point_possible_solution.zip`.

Only a constructor for Cartesian co-ordinates has been provided. One cannot directly provide a constructor for Cartesian co-ordinates **and** another constructor for polar co-ordinates, since they will have the same signature and will conflict. Can use the named constructor idiom if desired (e.g. see `http://www.parashift.com/c++-faq/named-ctor-idiom.html`).

Note that functions `set_x`, `set_y`, `set_r` and `set_theta` are provided independently (`set_x` preserves the current value of y and `set_r` preserves the current value of θ, etc) in addition to the functions `set_Cartesian` and `set_polar`.

The example solution stores only the Cartesian representation, and converts to and from polar co-ordinates when necessary. If you expect that your clients are more likely to be working in polar co-ordinates (i.e. more likely to call `get_r`, `get_theta`, `set_r`, `set_theta`, `set_polar`) then it may be more efficient to provide an implementation that stores only the polar co-ordinates instead. However, the interface of the class could be preseved in that case.

You could also provide an implementation that stored both the Cartesian and polar co-ordinates simultaneously. Then, the getters would be very simple (and quick), but the setters would always have to do some calculations to update both representations (to ensure consistency).

(c) How hard would it be to extend the functionality to points in 3-dimensional (or n-dimensional) space? (You don't necessarily have to try to do it.)

Three dimensions is conceptually easy, but would need to change most of the functions and interfaces and would involve considerable effort. Would need to replace polar coordinates with cylindrical coordinates, might want to add spherical polar co-ordinates, and might want to add projection support.

For n dimensions, would need to use vectors and consider generalising to hyper-spherical coordinates.

Beyond thinking about the interfaces, it is not terribly instructive to try to implement either, especially as we don't have any need for such classes at this time. (Anyone crazy enough to try to implement these may send me their solutions.)

Exercise 3: Linear interpolation example (extra exercise)

Another example of a class is provided in 09_Interp.zip. Note that the class has data members which are declared `const`, this means that they can only be set in the initialisation list in the constructor, and no member function can modify them.

(a) Look through the class and understand what it does.

(b) Data members of a class (like `x_` and `y_` here) can be references. Consider changing the line

```
const std::vector<double> x_, y_;
```

to

```
const std::vector<double> &x_, &y_;//to make two references, need two &s
```

or equivalently

```
const std::vector<double>& x_;
const std::vector<double>& y_;
```

. This would mean that the class holds a reference to the vectors which its constructor is given, rather than copies. What are the advantages and disadvantages of doing this? If the class does not have to copy its source data, then it will be faster and use less memory, which could be very advantageous.

The user of the class could change the contents of the x and y vectors during the life of the object, then subsequent calls to the methods would see the new version. The user would then have circumvented the error checking in the constructor of the class, which could make the assumptions made in the other methods invalid and lead to rubbish. The principle of encapsulation would have been violated.

(c) Implement the `get_many` function. Something like this

```
std::vector<double>
Interpolation::get_many(const std::vector<double>& new_x) const {
    std::vector<double> y;
    size_t index2 = 0;
    for(double d : new_x){
        for(;index2<x_.size();++index2){
            if(d<x_.at(index2)){
                break;
            }
        }
        if(index2==0){
            y.push_back(y_.front());
        }
        else if (index2==x_.size()){
            y.push_back(y.back());
        }
        else{
            auto index1 = index2-1u;
            double fraction = (d-x_.at(index1))/(x_.at(index2)-x_.at(index1));
            y.push_back(y_.at(index1) + fraction*(y_.at(index2)-y_.at(index1)));
        }
    }
    return y;
}
```

Remember to talk through your answers with the lab assistants!