DD2387 Programsystemkonstruktion med C++
Lab 2: The Essentials
10th of October 2015

Introduction

The purpose of this lab is to build on the elementary knowledge acquired in *lab1*. This includes, but is not limited to; the usage of inheritance, the keyword `virtual`, `friend`[1], and overall program design.

You are allowed to solve the assignments either by yourself, or with at most one (1) partner, where the latter is strongly advised. Do note that the oral presentation associated with this lab mandates that every member of your team is able to answer questions regarding every aspect of your implementations.

This means that even though you are allowed to divide the work load, knowledge of the entire implementation must be shared equally between both members.

Please read through the entire document prior to solving any of the problems listed, since most assignments are connected in one way or another.

General Requirements

- Your code should be modularized in classes and files, this includes the use of header files (`.hpp`) for declarations, and implementation files (`.cpp`) for the definitions.
- Make sure that your implementations are easy to read, maintain, and understand. This includes the usage of correct indentation, as well as having suitable names for your variables.
- Your implementations must not leak memory or any other acquired resource.

  Pay attention to the purpose of constructors, and their relationship with the corresponding destructor: *Resource Acquisition Is Initialization*[2].
- Your implementations should demonstrate that you fully understand the semantics associated with;
  - the keyword `const`,
  - the keyword `friend`, and;
  - class inheritance, including:
    - the keyword `virtual`,
    - the keyword `override`, and;
    - the semantics of access-levels (`private`, `protected`, and `public`).

Groundwork

***Note**: As long as you hold on to your lab report receipt, there's no need to go through these steps more than once.*

- Download and print the lab report receipt found at the course overview for DD2387 at KTH Social[3].
- Fill out the lab report receipt.

[1] ... but not foes `;-)`
[2] `http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization`
[3] `https://www.kth.se/social/course/DD2387/`

- Remember to ask for a signature after each oral presentation associated with a particular lab.

  All results are reported to `http://rapp.csc.kth.se`, and you are advised to check so that your results have been reported correctly. If results are missing in rapp you must contact the course leader and present your signed report receipt.

Submitting and presenting your work

Every assignment requires an oral presentation where you, and your potential partner, shall be able to explain and answer questions regarding your solutions.

Some assignments require you to submit code for automatic testing, which will verify that your implementation is correct in relation to the requirements set forth by the assignment in question.

To submit an implementation for automatic testing, open up a web browser and point it to `https://kth.kattis.com`, then;

- authenticate using your KTH-id, if this is the first time you are using *Kattis* you must register for the service after signing in, also;
- make sure that you register as a student taking cprog15, before you try to submit any of your solutions.

You are free to make as many submissions as you wish, but please note that the implementations you would like to present during the oral presentation must be submitted to, and be approved by, *Kattis*.

Additional Information

- When this document refers to files in the "*lab directory*", it is referring to the contents of `/info/DD2387/labs/lab2`, which can be accessed through `u-shell.csc.kth.se`.
- When this document refers to files in the "*assignment directory*", it is referring to a directory within the *lab directory*, that corresponds to the current assignment.

You may also find links to the relevant data by browsing the contents of:

- `http://www.csc.kth.se/utbildning/kth/kurser/DD2387/kurskatalog/`

Additional information, and the latest version of this document, is available at the course web:

- `https://www.kth.se/social/course/DD2387`.

Contents

0 The Essential (mandatory assignments)

A specific day in the history of the world can be represented as a date, but the "value" of this date depends on what representation we choose to use. For example; the 1st of August 2006 in the *Gregorian* calendar corresponds to the 19th of July 2006 in the *Julian* calendar.

The assignments in this lab all deal with dates in one way or another, and your task is to implement the different classes required to make the following, among other things, work as described.

```
{ Gregorian g (1900, 1,   1);
  Julian    j (1899, 12, 19);

  j == g; // shall yield false
  j++;    // increment ‘j‘ by one day
  j == g; // shall yield true
}

{ Gregorian g (1858, 11, 16);
  Julian    j (g);
  std::cout << j; // shall print 1858-11-4
  std::cout << g; // shall print 1858-11-16
}

{ Date * p1 = new Julian ();
  Date * p2 = new Gregorian ();

  *p1 == *p2; // shall be true

  delete p1;
  delete p2;
}
```

0.0 What is a `Date`?

`lab2::Date` shall be an abstract base-class (either directly or indirectly) of the date representation classes you are going to implement in this lab.

Every (non-abstract) class—`lab2::Gregorian` and `lab2::Julian`— that inherits from `lab2::Date` shall comply with the requirements in the upcoming sections.

General

- You shall apply correct usage of `const`, `friend`, `virtual`, and/or `override` where appropriate, even if the requirements does not explicitly state where these are to be used.
- The class shall, at least, be able to represent dates that are within the following range of years `[1858, 2558]`.
- If an operation yields a date that corresponds to a day outside the range of representation, an exception of type `std::out_of_range` shall be thrown.

Initialization and Assignment

- A *default-constructed* object shall contain a date that refers to the current day (i.e. the day when the program is executed).

  In order to obtain the current timestamp in a portable way (that also works on *Kattis*), see:

  - `http://j.mp/cprog15-kattistime`
- Besides support for *default-* and *copy-construction*, your class shall be able to be constructed;
  - using an object that inherits from `lab2::Date` (potentially using another date representation), and;
  - with three arguments of type `int` named; `year`, `month`, and `day` respectively.
    - If this constructor is given an invalid date, an exception of type `std::invalid_argument` shall be thrown.
- Besides support for *copy-assignment*, your class shall be able to be assigned a value;
  - from an object inheriting from `lab2::Date`.

    Note: The internal date representation shall not change, but the date itself might need a conversion since the given day potentially has another representation (in *source* vs *destination*).

Mandatory (`public`) *member-functions*

	Requirements
`int year()`	Returns the current year
`unsigned int month()`	Returns the number associated with the current month (`[1,n]`).
`unsigned int day()`	Returns the number associated with the current day in the current month (`[1,n]`).
`unsigned int week_day()`	Returns the number associated with the current weekday (`[1,n]`).
`unsigned int days_per_week()`	Returns the number of days in a week.
`unsigned int days_this_month()`	Returns the number of days in the current month.
`std::string week_day_name()`	Returns the name of the current weekday.
`std::string month_name()`	Returns the name of the current month.
`unspecified` `add_year (int n = 1)`	Increments the current year by `n`.
`unspecified` `add_month (int n = 1)`	Increments the current month by `n`.
`int mod_julian_day()`	This function shall return the *MJD* representation of the current date.

Note: If either `add_year` or `add_month` would yield an invalid date, the date shall automatically be adjusted to refer to the last day of the destination month.

Note: Calling `add_year` or `add_month` with a value of `1` N times might not yield the same result as invoking it once with N as argument.

```
// this pseudo-code uses the Gregorian calendar
2004-01-30.add_month (1) => 2004-02-29
2003-01-30.add_month (1) => 2003-02-28

2000-01-20.add_year  (5) => 2005-01-20
2004-02-29.add_year  (1) => 2005-02-28
2004-02-29.add_year  (4) => 2008-02-29
```

Mandatory operators

Sample Usage	**Description**
`++d` and `--d`	Shall increment/decrement the date by one day, and return a reference to the current object.
`d++` and `d--`	Shall increment/decrement the date by one day, and return a copy of the same type as `d`, having the value of `d` *prior* to any modification.
`d += n`	Shall increment the date by `int n` days. You shall decide upon a suitable return-type.
`d -= n`	Shall decrement the date by `int n` days. You shall decide upon a suitable return-type.
`a - b`	Shall yield the number days between the two dates. You shall decide upon a suitable return-type.
`a` *`relational-operator`* `b`	Where *relational-operator* is one of `==`, `!=`, `<`, `<=`, `>`, `>=`. All shall return an appropriate value of type *bool*.
`std::cout` `<< d1 << " " << d2`	It shall be possible to use `operator<<` to output the date-representation (as `year-month-day`, including the hyphens) on the `std::ostream&` (*`std::cout`* in the example).
`x = y`	Shall put the equivalent date represented by `y` in, `x`. **Example**: If `y` is `2006-08-01` *gregorian*, and `x` represents dates using the *julian* calendar; `x` shall contain `2006-07-19` after the assignment.

0.1 The Plan

Before you start working on the implementations required to fulfill the requirements listed in the previous section, you shall come up with an inheritance design.

You can come up with a design of your own, or choose from the below listed alternatives.

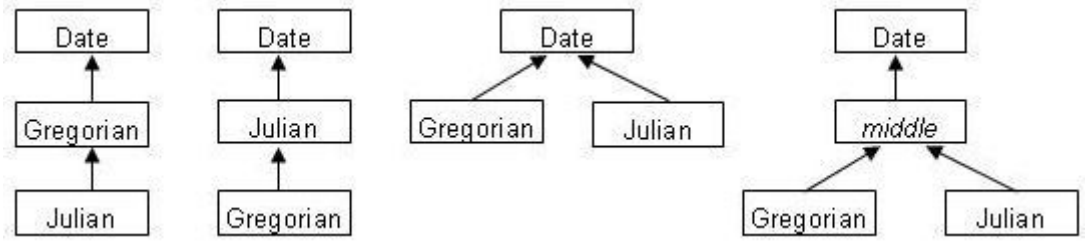

Figure 1: Design alternatives

0.1.1 Requirements

- You shall draw an inheritance tree that shows what you consider to be the most appropriate way to visualize, and implement, the relationship between `lab2::Date`, `lab2::Julian`, and `lab2::Gregorian`.
- You shall come up with *pros* and *cons* when comparing the designs presented in Figure 1 Design alternatives.
- If chosen design is not one of the previously listed, you shall come up with *pros* and *cons* related to the ones addressed in this document.

0.2 The Date

An abstract base-class is used to enforce that non-abstract classes that inherits from it implements a certain interface. Please remember that decisions made while designing this class will affect everything from this point forward.

0.2.1 Requirements

- Implement `lab2::Date`.
- Decide what constructors, if any, that shall be present in the *base-class*.
- Decide what *assignment-operator*s, if any, that shall be present in the *base-class*.

  Note: It shall be possible to use `*p1 = *p2` where both `p1` and `p2` are of type *pointer-to-date*.

  ```
  lab2::Date * p1 = new Julian ();
  lab2::Date * p2 = new Gregorian (1900, 1, 1);

  *p1 = *p2; std::cout << *p1; // 1899-12-20

  delete p2;
  delete p1;
  ```

- Member-functions that cannot be implemented directly in the base-class shall be made *pure virtual* to enforce that non-abstract children comply with the requirements.
- The member-functions that can be implemented but that inheriting classes might choose to `override` shall be declared `virtual`.
- The class declaration shall reside in a header named `date.hpp`.
- The class shall reside in `namespace lab2`.

0.2.2 Hints

- There is some functionality that can be neither declared nor implemented in the base-class due to the semantics regarding abstract classes.

0.2.3 Questions

- Why can prefix increment/decrement be declared in the abstract base-class, but not postfix?

0.3 Julius and Gregorius

As stated earlier in this document, a specific day in the history of time can be represented in different ways. Two — at least somewhat common — ways to *count and present* the past/future are the *Julian-* and *Gregorian calendar*.

Your task is to implement two classes, `lab2::Julian` and `lab2::Gregorian`, that makes it easy to deal with both of them without having to have detailed knowledge of how they work.

0.3.1 Requirements

- Implement `lab2::Julian` and `lab2::Gregorian`.
- The classes shall inherit from `lab2::Date` (either directly or indirectly).
- The requirements listed in *subsection 0.0 What is a `Date`?*.
- The class declarations shall reside in headers named `julian.hpp` and `gregorian.hpp`—respectively.
- Implementation-details—helper functions such as `is_leap_year()` and similar entities—shall be hidden from public view with `protected` or `private`.
- Your implementations (together with the code associated with `lab2::Date`) shall be uploaded to, and accepted by, *Kattis*.

0.3.2 Hints

- We are currently using the *Gregorian* calendar.
- Leap years are not handled the same way in the *Julian* vs *Gregorian* calendar.
- Most information necessary to complete this assignment can be found in the following linked article: `https://en.wikipedia.org/wiki/Julian_day#Calculation`
- The following link can be used to reach a *calculator* that converts between dates in different formats: `https://www.fourmilab.ch/documents/calendar/`

  Note: *If you input something in one of the forms and press "calculate", the result will be presented in the other forms.*
- You can test a small subset of the requirements in this lab by compiling your code together with `datetest.cpp` (available in the *lab directory*).

0.3.3 Some Additional Information

```
1/1   1900 Greg    = 20/12 1899 Jul
1/1   1900 Jul     = 13/1  1900 Greg
16/11 1858 Greg    = 4/11  1858 Jul   modified julian day -1
17/11 1858 Greg    = 5/11  1858 Jul   modified julian day  0
18/11 1858 Greg    = 6/11  1858 Jul   modified julian day  1
```

0.4 The Calendar

A calendar contains dates as well as events that are to occur (or has occurred) on those dates. In order to fully appreciate the work you have put into creating `lab2::Julian` and `lab2::Gregorian` you are to implement `template<typename DateType> class lab2::Calendar`.

0.4.1 Sample Usage

```
Calendar<Gregorian> cal;
cal.set_date(2000, 12, 2);
cal.add_event("Basketträning", 4, 12, 2000);
cal.add_event("Basketträning", 11, 12, 2000);
cal.add_event("Nyårsfrukost", 1, 1, 2001);
cal.add_event("Första advent", 1);          // år = 2000, månad = 12
cal.add_event("Vårdagjämning", 20, 3);      // år = 2000
cal.add_event("Julafton", 24, 12);
cal.add_event("Kalle Anka hälsar god jul", 24); // också på julafton
cal.add_event("Julafton", 24); // En likadan händelse samma datum ska
                               // ignoreras och inte sättas in i kalendern
cal.add_event("Min första cykel", 20, 12, 2000);
cal.remove_event("Basketträning", 4);

std::cout << cal; // OBS! Vårdagjämning och första advent är
                  // före nuvarande datum och skrivs inte ut
std::cout << "----------------------------------------" << std::endl;
cal.remove_event("Vårdagjämning", 20, 3, 2000);
cal.remove_event("Kalle Anka hälsar god jul", 24, 12, 2000);
cal.set_date(2000, 11, 2);
if (! cal.remove_event("Julafton", 24)) {
     std::cout << " cal.remove_event(\"Julafton\", 24) tar inte"<< std::endl
               << " bort något eftersom aktuell månad är november" << std::endl;
}
std::cout << "----------------------------------------" << std::endl;
std::cout << cal;
```

0.4.2 Requirements

- The class template shall accept a `DateType` as its first *template-parameter*, which denotes the type used for the representing dates.
- The class template shall internally have a list of events (such as "Julafton", "Födelsedag", etc) associated with the relevant date.
- The class template shall have the following (`public`) constructors:
  - a default-constructor that sets the current calendar date to todays date,
  - suitable constructor(s) so that one can initialize the `Calendar` with another `Calendar` (potentially instantiated with another date-type).
- It shall be possible to print the events with their corresponding dates in a calendar by using *`std::cout`* `<< calendar` (where *`std::cout`* could be any `std::ostream&`).
  - The output shall be in the *iCalendar* format, make sure that it is actually parsable by some device (such as your phone).
  - Only events on the current date of the calendar, and future events, shall be included in the output.
- **public member-functions**

	Requirements
`bool set_date(int y, int m, int d)`	Sets the internal date. If the date is invalid, `false` shall be returned, otherwise `true`.
`bool add_event (...)`	Should accept 1 to 4 parameters; – `std::string event_name` – `int day` – `int month` – `int year` If trailing arguments are not passed when invoking the function, the missing information is to be pulled from the internal date of the calendar. If the call results in an invalid date, or if the *event description* is already present at the desired date, `false` shall be returned, otherwise `true`.
`bool remove_event (...)`	Same parameters as `add_event`. If there is no such corresponding event, the function shall return `false`, otherwise `true`.

0.4.3 Hints

- You are allowed, and encouraged, to use the containers available in the Standard Library in order to solve this assignment.
- `https://en.wikipedia.org/wiki/ICalendar`
- The following link can be used to reach a tool that verifies data in the iCal format: `http://severinghaus.org/projects/icv/`