Class design

Constructing a date class

```
Date today;
Date d1(25, 12, 2007);
Date d2(12, 25, 2007);

Date d3 = d1 + 3;          // OK
int daysGone = d3 – d1; // OK

d1 + d2              // ??
(d1 + 5) – d3     // ??
d1.addyear(3);    // ??
d1.addmonth(2);  // ??
Date d(13);        // ??
```

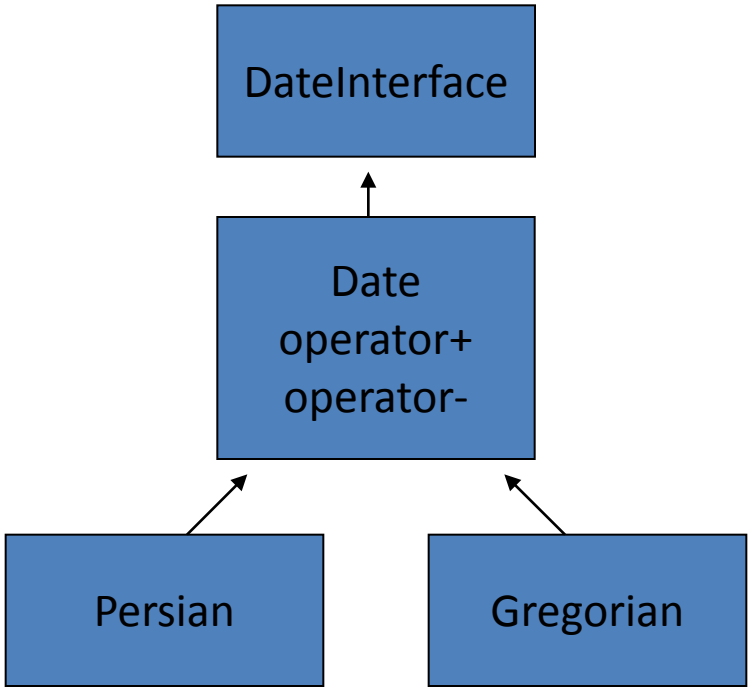
Class design

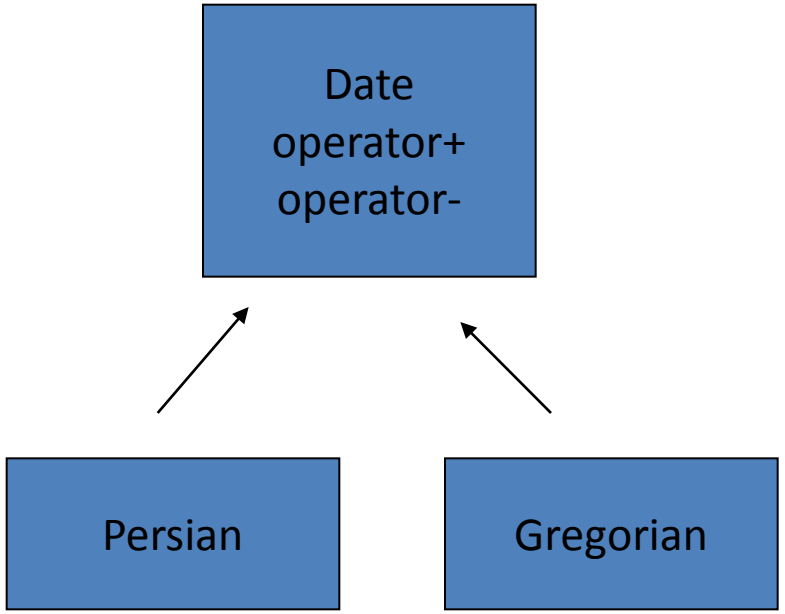
- The functions should be reasonable
- Allowed operations should make sense
- Can I use inheritance?
  - Chinese d2; // today
  - Persian d1; // today
  - int zero = d1 – d2;
  - Gregorian d3 = d2 + 10;

Date as base class

- Recap – make the destructor virtual?
- There is no dynamic memory
- Rule of three does not apply
- What do we know of future subclasses?

How to inherit?





Where do I put the code?

How to construct a date

Suppose we allow the following constructions

```
Date d3;        // today
Date d4(5, 3); // 5/3 current year
Date d5(25);   // 25th current month

Date(int d=0, int m=0, int y=0) {…
  //  Why is this constructor a bad idea?
```

Compile time vs runtime

- Do you prefer compile time error or runtime error?

```
Date d(Day(1), Month(2), Year(2003));
Date d(Month(2), Day(1),, Year(2003));

Date d(Day(1), Month::Jan(), Year(2007));
```

```
struct Month {
  static Month Jan() { return Month(1); }
  static Month Jan() { return Month(1); }
…

private:
  explicit Month(int x) : month(x) {}
  int month;
}
```

- static methods to make sure it is initialized
- Explicit constructor to avoid implicit conversions

Always consider explicit on one-argument constructors!

Compile time vs runtime

```
Date d(1, 1, 2007);
Date d(Day(1), Month::Jan(), Year(2007));
```

- What are the benefits?
- The first one is easier to type
- The first one is *seamingly* easier to guess
- The second one catches errors in compile time !!
- Always prefer compile time error checking

Exposing data

- Sometimes a struct is a struct

```
struct coord {
  float x;  float y; };
```

- Sometimes you need to change it

```
struct coord {
  double x;  double y; };
```

- Sometimes it's not what you think

```
struct coord {
  double q1;  double q2; }; // radius, fi
```

Exposing data

- Prefer d.getDay() instead of d.day
- Exposing day, month, year **commits**!
- We cannot change internal representation!
- Probably easier to represent a date as number of days from a fixed date
  - 1970-01-01, 1900-01-01, Julian day
- makes comparing persian and gregorian easy

Encapsulate data

- Hide data from other programmers
- Let them access via accessors/inspectors
- Let them change via mutator functions
- Incedently that's one of the cornerstones of object oriented programming

- Another cornerstone being organizing the code, put functions that handle the data together with the data (in the same class)

Class design of a calendar

```
Calendar c;
c.setdate(24, 12, 2007);
c.add_event(”Buy gifts, if you haven’t
  already!!!”);

c.add_event(”Meeting”, 2007, 5); // when is
                                 // that?

void setdate(int d = -1, m = -1, y = -1) {…
```

Still a very bad idea!

Why a bad idea

- The programmer did not have to explicitly write 4 functions (12 code lines?)
- He probably saved 6 minutes (2 lines/minute)
- The debugging programmer is looking for a valid date that shouldn’t be in the calendar in runtime …
- probably had to spend more than 6 minutes looking for the cause

Exercise - Design a calendar class

- How to construct the class?
- What operations is allowed (+ - * etc)?
- What functions do I want?
  - Compare with what I can do with a simple calendar paperback booklet?
- How do I handle a Persian calendar?
  - Can I convert my scheduled meetings in Iran?
  - Can I merge persian and chinese appointments?

Generic programming

- Sometimes your functions apply to more than one type
- What do you need to know when sorting a container?
- You need to know how to compare the items inside
- STL uses iterators to point into containers. std::sort assumes operator< is implemented
- If it is not implemented we get a **compile** error

Generic programming benefits

- Reusing code
  - one sort implemetation
- Compile time checking
  - operator< missing
- Optimization in runtime
  - Can optimize containers for simple types

Generic programming

- Instantiate your calendar with the type it should handle
  - Calendar<Persian> persian_calendar;
  - Calendar<Gregorian> gregorian_calendar;
  - gregorian_calendar += persian_calendar; // !!

Dependencies

Event

```
Event e (Date (Day(1), Month::Jan(),
  Year(2008)), ”Happy new year”);
```

- Let’s define an event consisting of a date and some description string.
- How shall I store the string and the date internally?

On the stack

As an instance member the date will be allocated on the stack

It might improve performance

```
class Event {
  Date date;
  string description;
  …
}
```

Compile dependancy

- Constructing the date object on the stack requires knowledge in compile time. How big is the object to put on the stack?
- It requires a compilation dependancy. Build time will increase
- An alternative is to use a pointer

Compile time dependancies

```
class Event {
  auto_ptr<Date> date;
   …
```

- To remove the compile dependancy make the member variable a pointer
- A pointer can always be allocated on the stack without knowing the size of the object it points to.

Performance

- The actual date object will be on the heap
- It might take a while to fetch it in runtime
- On the other hand it might be cached

Do not make decisions on assumptions

Test performance with a profiler

Until next time

- Object oriented programming
  - Important concepts
    - is a
    - has a
- Object modelling
  - Several solutions possible
  - fuzzy (flummigt) subject, sometimes religous
- Homework
  - Apply ”is a” and ”has a” on Date, Calendar, Event