`dated<T>`, a template for elements with accompanying dates

Bernt Arne Ødegaard

April 2007

Chapter 1

Dated

1.1 Introduction

A convenient data structure is a mapping of dates with some variable. For example, a time series of economic variables. The mapping is assumed to be one–to–one.

1.2 Implementation

This is implemented as a template class, where the data is two vectors. One of type date, the other of type `<T>`, the user defined type.

1.3 User functions

- Adding data: `insert, append`
- Removing data: `clear, remove, remove_between, remove_after, remove_before...`
- Querying: `contains, first_date, last_date`
- Picking data: `date_at, element_at, dates(), elements()`

```
#ifndef _DATED_H_
#define _DATED_H_

#include <vector>
#include "date.h"   // my date class

template <class T> class dated {
private:
    vector<date> dates_;
    vector<T> elements_;
public:
    dated<T>();
    dated<T>(const dated<T>&);
    dated<T> operator= (const dated<T>&);
    ~dated() { clear(); };
    void clear();                    // erasing
    void insert(const date&, const T&); // insert somewhere

    bool empty() const ;
    int size() const ;
    bool contains(const date& d) const ;
    date date_at(const int& t) const ; // accessing elements, here dates
    T element_at(const int& t) const ; // index directly
    T element_at(const date& d) const ; // index indirectly, specify what date

    // next: the element either on date d, if d is here, else the last observation before d.
    T current_element_at(const date& d) const ;

    vector<T> elements() const; // all elements as vector<T>
    vector<date> dates() const; // all dates as vector<date>

    date first_date() const; // simple queries
    date last_date() const;
    T first_element() const;
    T last_element() const;

    int index_of_date(const date& d) const; // when searching in the data,
    int index_of_last_date_before(const date& d) const; // these are useful functions
    int index_of_first_date_after(const date& d) const;

    void remove(const date&);   // removing one or more elements
    void remove_between_including_end_points(const date&, const date&);
    void remove_between(const date&, const date&);
    void remove_before(const date&);
    void remove_after(const date&);
};

#include "dated_main.h"
#include "dated_search.h"
#include "dated_remove.h"

template<class T> dated<T> observations_between(const dated<T>& obs,const date&first, const date& last);
template<class T> dated<T> observations_after(const dated<T>& obs, const date& first);
template<class T> dated<T> observations_before(const dated<T>& obs, const date& last);
template<class T> dated<T> end_of_year_observations(const dated<T>&);
template<class T> dated<T> beginning_of_month_observations(const dated<T>&);
template<class T> dated<T> end_of_month_observations(const dated<T>&);
template<class T> dated<T> observations_matching_dates(const dated<T>& obs, const vector<date>& dates);

#include "dated_util.h"
#endif
```

Header file 1.1: dated h

```
template<class T> dated<T>::dated(){;}; // not necessary to do anything,

template<class T>dated<T>::dated(const dated<T>& dobs) {
    // for speed, initialize first with correct size and then copy
    dates_=vector<date>(dobs.size());
    elements_=vector<T>(dobs.size());
    for (unsigned int t=0;t<dobs.size();++t){
        dates_[t]      = dobs.date_at(t);
        elements_[t]  = dobs.element_at(t);
    };
};

template<class T> dated<T> dated<T>::operator= (const dated<T>& dobs) {
    if (this==&dobs) return *this; // check against self assignment;
    clear();
    dates_=vector<date>(dobs.size());
    elements_=vector<T>(dobs.size());
    for (unsigned int t=0;t<dobs.size();++t){
        dates_[t]       = dobs.date_at(t);
        elements_[t]    = dobs.element_at(t);
    };
    return *this;
};

template<class T> dated<T>::~dated();

template<class T> bool dated<T>::empty() const { return (dates_.size()<1); };

template<class T> int dated<T>::size() const { return int(dates_.size()); };

template<class T> date dated<T>::date_at(const int& t) const { // accessing with bounds checking
   if ( (t>=0) && (t<size()) ) return dates_[t];
   return date();
};

template<class T> T dated<T>::element_at(const int& t) const { // accessing with bounds checking
   if ( (t>=0) && (t<size()) ) return elements_[t];
   return T();
};

template<class T> T dated<T>::element_at(const date& d) const {
   if (!contains(d)) return T();
   return elements_[index_of_date(d)];
};

template<class T> T dated<T>::current_element_at(const date& d) const {
    // the element either on date d, if d is here, else the last observation before d.
    if (size()<1) return T();
    if (contains(d)) return element_at(d);
    if (d<first_date()) { return T(); };
    return elements_[index_of_last_date_before(d)];
};

template<class T> vector<T> dated<T>::elements() const {
    vector<T> elements(size());
    for (unsigned int t=0; t<size(); ++t){ elements[t]=element_at(t); };
    return elements;
};;

template<class T> vector<date> dated<T>::dates() const {
    vector<date> ds(size());
    for (unsigned int t=0;t<size();++t){ ds[t]=date_at(t); };
    return ds;
};

template <class T> void dated<T>::insert(const date& d, const T& obs) {
    if (!d.valid()) return;
    if ( (empty() ) || (d>last_date()) ) {
        dates_.push_back(d);
        elements_.push_back(obs);
        return;
    };
    if (d<first_date()) {
        dates_.insert(dates_.begin(),d);
        elements_.insert(elements_.begin(),obs);
```

```
#include <algorithm>

template<class T> bool dated<T>::contains(const date& d) const {
   return binary_search(dates_.begin(),dates_.end(),d);
};

template<class T> date dated<T>::first_date() const {
    if (empty()) return date();
    return dates_.front();
};

template<class T> date dated<T>::last_date() const {
   if (empty()) return date();
   return dates_.back();
};

template<class T> T dated<T>::first_element() const {
   if (empty()) return T();
   return elements_.front();
};

template<class T> T dated<T>::last_element() const {
   if (empty()) return T();
   return elements_.back();
};

template <class T> int dated<T>::index_of_date(const date& d) const {
   // this routine returns the index at which date d is, or -1 if not found.
   if (!d.valid()) return -1;
   if (!contains(d)) return -1;
   int dist=0;
   for (unsigned int i=0;i<dates_.size();++i){
       if (dates_[i]==d) return i; // slow implementation, but works (for now),
   };
   return dist;
};

template <class T> int dated<T>::index_of_first_date_after(const date& d) const {
    // this routine returns the index of the first date after d.
    if (!d.valid()) return -1;
    if (d>=last_date()) return -1;
    if (d<first_date()) return 0;
    for (unsigned int i=0;i<dates_.size();++i){
        if (dates_[i]>d) return i;
    };
    return -1;
};

template <class T> int dated<T>::index_of_last_date_before(const date& d) const {
    // this routine returns the index of the first date before d.
    if (!d.valid()) return -1;
    if (d<=first_date()) return -1;
    if (d>last_date()) return index_of_date(last_date());
    for (unsigned int i=0;i<dates_.size();++i){
        if (dates_[i]>=d) return i-1; // slow implementation, but works (for now)
    };
    return -1;
};
```

Header file 1.3: Searching

```
template<class T> void dated<T>::clear() {
   dates_    .erase(dates_    .begin(),dates_    .end());
   elements_ .erase(elements_ .begin(),elements_ .end());
};

template<class T> void dated<T>::remove(const date& d) {
   int i=index_of_date(d);
   if (i>=0) {
      dates_.erase(dates_.begin()+i);
      elements_.erase(elements_.begin()+i);
   };
};

template<class T> void dated<T>::remove_between(const date& d1, const date& d2) {
    //    cout << " removing between " << d1 << d2 << endl;
    if (!d1.valid()) return;
    if (!d2.valid()) return;
    if ( (d1<first_date()) && (d2>last_date()) ) {
         dates_.clear();
         elements_.clear();
         return;
    };
    /* below has a bug, use the slow version for now
    ** problem is that the last observation before the one to be removed is not removed.
       int first=index_of_first_date_after(d1);
       int last=index_of_last_date_before(d2);
       cout << " first " << first << " last " << last << endl;
       cout << " before " << first_date() << last_date() << endl;
       if ( (first>=0) && (last>=0) ) {
       if (first==last) { // just remove one element
       dates_ .erase(dates_ .begin()+first);
       elements_ .erase(elements_ .begin()+first);
       }
       else if (first<last) {
       if (d2>last_date()){
       dates_ .erase(dates_ .begin()+first,dates_ .end());
       elements_ .erase(elements_ .begin()+first,elements_ .end());
       }
       else if (d1<first_date()) {
       dates_ .erase(dates_ .begin(),dates_ .begin()+last);
       elements_ .erase(elements_ .begin(),elements_ .begin()+last);
       }
       else {
       dates_ .erase(dates_ .begin()+first,dates_ .begin()+last);
       elements_ .erase(elements_ .begin()+first,elements_ .begin()+last);
       };
       };
       };
       cout << " after " << first_date() << last_date() << endl;
    */

    for (int t=size()-1;t>=0;t--){ // this is very slow, to be replaced with one using vector erase.
         date d=date_at(t);
         if ( (d>d1) && (d<d2) ) {
              dates_.erase(dates_.begin()+t);
              elements_.erase(elements_.begin()+t);
         };
    };

};


template<class T> void dated<T>::remove_between_including_end_points(const date& d1, const date& d2) {
    if (!d1.valid()) return;
    if (!d2.valid()) return;
    remove_between(d1,d2); // simply use above, and then remove two end points
    remove(d1);
    remove(d2);
    /*
    for (int t=size()-1;t>=0;t--){ // this is very slow, to be replaced with one using vector erase.
         date d=date_at(t);
         if ( (d>=d1) && (d<=d2) ) {
              dates_.erase(dates_.begin()+t);
              elements_.erase(elements_.begin()+t);
```

```
#ifndef _DATED_UTIL_H_
#define _DATED_UTIL_H_

template<class T> dated<T> observations_between( const dated<T>& obs, const date& first, const date& last) {
    //    cout << " picking obs between " << first << last << endl;
    dated<T> picked = obs; // assume that the first and last date should be included.
    picked.remove_after(last);// just copy and then remove. Fast enough
    picked.remove_before(first);
    return picked;
};

template<class T> dated<T> observations_after( const dated<T>& obs, const date& first) {
    // assume that the first date is to be included in the result // should maybe be observations_on_and_after....
    dated<T> dobs = obs; // just copy and then remove. Fast enough
    dobs.remove_before(first);
    return dobs;
};

template<class T> dated<T> observations_before( const dated<T>& obs, const date& last) {
    dated<T> dobs = obs; // assume that the last date is to be included in the result
    dobs.remove_after(last);
    return dobs;
};

template<class T> dated<T> end_of_year_observations(const dated<T>& dobs) {
    dated<T> eoy_obs;
    if (dobs.first_date().month()==1) {// take first obs in january as end of previous year
        eoy_obs.append(dobs.date_at(0),dobs.element_at(0));
    };
    for (unsigned int t=0;t<dobs.size()-1;++t) {
        if (dobs.date_at(t).year()!=dobs.date_at(t+1).year()) {
            eoy_obs.append(dobs.date_at(t),dobs.element_at(t));
        };
    };
    if (eoy_obs.last_date().year() != dobs.last_date().year()) {
        eoy_obs.append(dobs.last_date(),dobs.element_at(dobs.size()-1));
    }
    return eoy_obs;
}

template<class T> dated<T> beginning_of_month_observations(const dated<T>& dobs) {
    dated<T> eom_obs;
    eom_obs.append(dobs.date_at(0),dobs.element_at(0));// take first observation always
    for (unsigned int t=1;t<dobs.size();++t) {
        if ( (dobs.date_at(t).month()!=dobs.date_at(t-1).month())|| (dobs.date_at(t).year()!=dobs.date_at(t-1).year()) ) {
            eom_obs.append(dobs.date_at(t),dobs.element_at(t));
        };
    };
    return eom_obs;
}

template<class T> dated<T> end_of_month_observations(const dated<T>& dobs) {
   dated<T> eom_obs;
   for (unsigned int t=0;t<dobs.size()-1;++t) {
      if ( (dobs.date_at(t).month()!=dobs.date_at(t+1).month()) || (dobs.date_at(t).year()!=dobs.date_at(t+1).year())) {
          eom_obs.append(dobs.date_at(t),dobs.element_at(t));
      };
   };
   if ( (eom_obs.last_date().month() != dobs.last_date().month()) || (eom_obs.last_date().year() != dobs.last_date().year()) ) {
      eom_obs.append(dobs.last_date(),dobs.element_at(dobs.size()-1));
   }
   return eom_obs;
};

template<class T> dated<T> observations_matching_dates( const dated<T>& obs, const vector<date>& dates){
   dated<T> dobs;
   for (unsigned int t=0;t<dates.size();++t){
      if (obs.contains(dates[t])) {
          dobs.append(dates[t],obs.element_at(dates[t]));
      };
   };   return dobs;
};
#endif
```


Header file 1.5: Utilities (nonmember functions)