C++ classes for empirical financial data

Bernt Arne Ødegaard[1]

April 2007

[1]Norwegian School of Management BI and Norges Bank

Chapter 1

Introduction

In this document we describe a self-contained set of utilities that should be of use for academics (and practitioners) working with financial data.

Chapter 2

Date

This is the documentation for a C++ date class.

The date class is pretty rough. A date is stored as three integers (year, month, day). Functions for comparing dates, incrementing dates are provided. Also calculation of time between dates.

2.1 Setting up.

Compile date.cc, put into library libdate.a, make library available.

Put date.h on header file path.

2.2 Defining dates.

Examples of date definitions

`date d;`

`date d(19930624)`

`date d(24,6,1993)` Note the day, month, year sequence.

2.3 Operations on dates.

Let `d, d1, d2` be dates.

The following operations are defined:

Iteration: `d++`, `++d`, `d--`, `--d` .

Logical comparisions >, >=, <, <=, ! =, ==.

Some aritmetric operations are defined, which has a meaning that may seem strange at first, but they are natural for usage purposes.

Adding/Subtracting x number of days from a data:

`d=d1+7` gives `d` the date 7 days after date `d1`.

`d=d1-7` gives `d` the date 7 days before date `d1`.

Finding the number of days between 2 dates. `i=d1-d2`. Here `i` is an integer that is the number of days between `d1` and `d2`. `i` will be negative if `d1` $<$ `d2`.

Example:

```
date d1(30,6,1993); date d2(19930610);
i = d1-d2
```

Gives `i` the value 20, the number of days between 10 jun 93 and 30 jun 93.

```
// file: date.h
// author: Bernt A Oedegaard.

#ifndef _DATE_H_
#define _DATE_H_

#include <iostream>
using namespace std;

class date {
protected:
    int year_;
    int month_;
    int day_;
public:
    date();
    date(const int& d, const int& m, const int& y);

    bool valid(void) const;

    int day()  const;
    int month() const;
    int year() const;

    void set_day (const int& day );
    void set_month (const int& month );
    void set_year (const int& year );

    date operator ++(); // prefix
    date operator ++(int); // postfix
    date operator --(); // prefix
    date operator --(int); // postfix
};

bool operator == (const date&, const date&); // comparison operators
bool operator != (const date&, const date&);
bool operator < (const date&, const date&);
bool operator > (const date&, const date&);
bool operator <= (const date&, const date&);
bool operator >= (const date&, const date&);

ostream& operator << ( ostream& os, const date& d); // output operator

#endif
```

Header file 2.1: Header file

```
#include "date.h"

////////////////////////////// construction //////////

date::date(const int& d, const int& m, const int& y) {
   day_  = d;
   month_ = m;
   year_ = y; // this assumes year is given fully, not Y2K corrections
};

////////////////////////////// inline definitions //////////

date::date(){ year_ = 0; month_ = 0; day_ = 0;};

int date::day()  const { return day_; };
int date::month() const { return month_; };
int date::year() const { return year_; };

void date::set_day (const int& day) { date::day_ = day; };
void date::set_month(const int& month) { date::month_ = month; };
void date::set_year (const int& year) { date::year_ = year; };

bool date::valid() const {
// This function will check the given date is valid or not.
// If the date is not valid then it will return the value false.
//  Need some more checks on the year, though
    if (year_ <0)               return false;
    if (month_>12 || month_<1) return false;
    if (day_>31 || day_<1)   return false;
    if ((day_==31 &&
         ( month_==2 || month_==4 || month_==6 || month_==9 || month_==11) ) )
        return false;
    if ( day_==30 && month_==2) return false;
    if ( year_<2000){
        if ((day_==29 && month_==2) && !((year_-1900)%4==0)) return false;
    };
    if ( year_>2000){
        if ((day_==29 && month_==2) && !((year_-2000)%4==0)) return false;
    };
    return true;

};
```

C++ Code 2.1: Defining the basic operations

```
#include "date.h"

bool operator == (const date& d1,const date& d2){
// check for equality
    if (!d1.valid()) { return false; };
    if (!d2.valid()) { return false; };
    if( (d1.day()==d2.day())
        && (d1.month()==d2.month())
        && (d1.year()==d2.year())) {
        return true;
    };
    return false;
}

bool operator !=(const date& d1, const date& d2){
    return !(d1==d2);
}

bool operator < (const date& d1, const date& d2){
    if (!d1.valid()) { return false; }; // not meaningful, return anything
    if (!d2.valid()) { return false; }; // should really be an exception, but ?
    if (d1.year()<d2.year())    { return true;}
    else if (d1.year()>d2.year()) { return false;}
    else {                              // same year
        if (d1.month()<d2.month()) { return true;}
        else if (d1.month()>d2.month()) { return false;}
        else { // same month
            if ( d1.day()<d2.day()) { return true;}
            else { return false; }
        };
    };
    return false;
};

bool operator > (const date& d1, const date& d2) {
    if (d1==d2) { return false;}; // this is strict inequality
    if (d1<d2) { return false; };
    return true;
}

bool operator <=(const date& d1, const date& d2){
  if (d1==d2) { return true; }
  return (d1<d2);
}

bool operator >=(const date& d1, const date& d2) {
    if (d1==d2) { return true;};
    return (d1>d2);
};
```

C++ Code 2.2: Comparisons

```
#include "date.h"

inline date next_date(const date& d){
    date ndat;
    if (!d.valid()) { return ndat; };
    ndat=date((d.day()+1),d.month(),d.year()); if (ndat.valid()) return ndat;
    ndat=date(1,(d.month()+1),d.year()); if (ndat.valid()) return ndat;
    ndat = date(1,1,(d.year()+1));       return ndat;
}

inline date previous_date(const date& d){
    date ndat;
    if (!d.valid()) { return ndat; }; // return zero
    ndat = date((d.day()-1),d.month(),d.year()); if (ndat.valid()) return ndat;
    ndat = date(31,(d.month()-1),d.year()); if (ndat.valid()) return ndat;
    ndat = date(30,(d.month()-1),d.year()); if (ndat.valid()) return ndat;
    ndat = date(29,(d.month()-1),d.year()); if (ndat.valid()) return ndat;
    ndat = date(28,(d.month()-1),d.year()); if (ndat.valid()) return ndat;
    ndat = date(31,12,(d.year()-1));       return ndat;
};

date date::operator ++(int){ // postfix operator
   date d = *this;
   *this = next_date(d);
   return d;
}

date date::operator ++(){ // prefix operator
   *this = next_date(*this);
   return *this;
}

date date::operator --(int){ // postfix operator, return current value
   date d = *this;
   *this = previous_date(*this);
   return d;
}

date date::operator --(){ // prefix operator, return new value
   *this = previous_date(*this);
   return *this;
};

inline long long_date(const date& d) {
    if (d.valid()){ return d.year() * 10000 + d.month() * 100 + d.day(); };
    return -1;
};

ostream & operator << (ostream& os, const date& d){
    if (d.valid()) {       os << " " << long_date(d) << " " ; }
    else {       os << " invalid date "; };
    return os;
}
```

C++ Code 2.3: Iteration

Chapter 3

Dated

3.1 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.

3.2 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.

3.3 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 3.1: 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 3.3: 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 3.5: Utilities (nonmember functions)

Chapter 4

Dated observations

A specialization of the general `dated<T>` class to be used for time series of numbers.

```
#ifndef _DATED_OBS_H_
#define _DATED_OBS_H_

#include "dated.h" // templated dated<> class
#include <string> // ANSI string class

class dated_observations : public dated<double>{
   private:
      string          title_;
   public:
      ~dated_observations() { clear(); };
      void clear();

      void set_title(string s); // title
      string title() const;

      int no_obs() const;
      int no_obs_between(const date& d1, const date& d2) const;
      int no_obs_before(const date& d) const;
      int no_obs_after(const date& d) const;
};

///// io
ostream& operator << (ostream&, const dated_observations& );
void print_dated_observations(ostream& of, const dated_observations& d, int precision=3);

//// mischellaneous utilities
double max_obs(dated_observations& dobs);
double min_obs(dated_observations& dobs);
bool dates_match(const dated_observations& obs1, const dated_observations& obs2);

///// picking subsets.
dated_observations observations_between(const dated_observations& obs,const date& first,const date& last);
dated_observations observations_after(const dated_observations& obs, const date& first);
dated_observations observations_before(const dated_observations& obs, const date& last);
dated_observations observations_matching_dates(const dated_observations& obs, const vector<date>& dates);

///// picking periodic elements
dated_observations beginning_of_month_observations(const dated_observations&);
dated_observations end_of_month_observations(const dated_observations&);
dated_observations end_of_year_observations(const dated_observations&);
dated_observations end_of_year_current_observations(const dated_observations&);

#endif
```

Header file 4.1: dated obs h

```
#include "dated_obs.h"
#include <algorithm>

void dated_observations::set_title(string s) { title_ = s; };

void dated_observations::clear() {
   title_ = string();
   dated<double>::clear();
};

string dated_observations::title() const { return title_;};

int dated_observations::no_obs_between(const date& dat1, const date& dat2) const {
// count number of observations between given dates.
   if (!dat1.valid()) return -1;
   if (!dat2.valid()) return -1;
   int noobs=0;
   int T = size();
   for (int t=0;t<T;++t) {
      if ( (dat1<=date_at(t) ) && (date_at(t)<=dat2) ) ++noobs;
   };
   return noobs;
};

int dated_observations::no_obs_before(const date& d) const {
   return no_obs_between(first_date(),d);
};

int dated_observations::no_obs_after(const date& d) const {
   return no_obs_between(d,last_date());
};
```

C++ Code 4.1: dated obs cc

```
#include "dated_obs.h"

bool dates_match(const dated_observations& d1, const dated_observations& d2){
   if (d1.size()!=d2.size()) { return false; }
   for (unsigned int t=0;t<d1.size();++t){
      // slow, careful check that the time series match exactly
      if (d1.date_at(t) != d2.date_at(t)) return false;
   };
   return true;
};

double max_obs(dated_observations& dobs){
   vector<double> obs = dobs.elements();
   return *max_element(obs.begin(),obs.end());
};


double min_obs(dated_observations& dobs){
   vector<double> obs = dobs.elements();
   return *min_element(obs.begin(),obs.end());
};
```

C++ Code 4.2: calc

```
#include "dated_obs.h"
#include <iomanip>

void print_dated_observations(ostream& of,
                              const dated_observations& obs,
                              int precision) {
   if (obs.title().length()>0) of << obs.title() << endl;
   for (unsigned int t=0;t<obs.size();t++) {
       of << obs.date_at(t) << " "
          << setprecision(precision) << fixed << obs.element_at(t)
          << endl;
   };
};

ostream& operator << (ostream& os, const dated_observations& obs) {
    print_dated_observations(os,obs,4);
    return os;
};
```

C++ Code 4.3: io

```
#include "dated_obs.h"

dated_observations end_of_year_observations(const dated_observations& dobs) {
   if (dobs.size()<1) return dated_observations();
    dated_observations eoy_obs;
    eoy_obs.set_title(dobs.title());
    if (dobs.first_date().month()==1) { // include beginning of first year
        eoy_obs.insert(dobs.date_at(0),dobs.element_at(0));
    };
    for (int t=0;t<dobs.size()-1;++t) {
        if (dobs.date_at(t).year()!=dobs.date_at(t+1).year()) {
            eoy_obs.insert(dobs.date_at(t),dobs.element_at(t));
        };
    };
    if (eoy_obs.last_date().year() != dobs.last_date().year()) {
        eoy_obs.insert(dobs.last_date(),dobs.element_at(dobs.size()-1));
    }
    return eoy_obs;
};

dated_observations end_of_year_current_observations(const dated_observations& dobs) {
   if (dobs.size()<1) return dated_observations();
    dated_observations eoy_obs;
    eoy_obs.set_title(dobs.title());
    // may want to keep the first observation if it is in january, and pub it in as last december,
    // but, for currently do not do that
    //    if (dobs.first_date().month()==1) {      eoy_obs.insert(date(31,12,first_date.year()-1),dobs.element_at(0)); };
    for (int year=dobs.first_date().year(); year<=dobs.last_date().year(); ++year){
        eoy_obs.insert(date(31,12,year),dobs.current_element_at(date(31,12,year)));
    };
    return eoy_obs;
}

dated_observations beginning_of_month_observations(const dated_observations& dobs) {
    if (dobs.size()<1) return dated_observations();
    dated_observations eom_obs;
    eom_obs.set_title(dobs.title());
    if (dobs.first_date().day()<5) { eom_obs.insert(dobs.date_at(0),dobs.element_at(0)); };
    for (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.insert(dobs.date_at(t),dobs.element_at(t));
        };
    };
    return eom_obs;
}

dated_observations end_of_month_observations(const dated_observations& dobs) {
    if (dobs.size()<1) return dated_observations();
    dated_observations eom_obs;
    eom_obs.set_title(dobs.title());
    if (dobs.first_date().day()<10) { eom_obs.insert(dobs.date_at(0),dobs.element_at(0)); };
    for (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.insert(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.insert(dobs.last_date(),dobs.element_at(dobs.size()-1));
    }
    return eom_obs;
};
```

C++ Code 4.4: Periodic

```
#include "dated_obs.h"

dated_observations observations_between( const dated_observations& obs,
                                         const date& first,
                                         const date& last) {
// assume that the first and last date should be included.
    dated_observations picked = obs; // just copy and then remove. Fast enough
    picked.remove_after(last);
    picked.remove_before(first);
    return picked;
};

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

dated_observations observations_before( const dated_observations& obs,
                                        const date& last) {
// assume that the last date is to be included in the result
   dated_observations dobs = obs;
   dobs.remove_after(last);
   return dobs;
};
```

C++ Code 4.5: Subsets

Chapter 5

Security price history

The following class is used when we have a lot of historical data about securities. Given price observations, do various calculations and pulling of data, as well as printing various reports.

5.0.1 Security price history

When doing empirical work in finance, we always end up with the same basic problem. From a time series of price observations of a financial security, we want to calculate any number of things: Returns at different frequencies, average returns, max, min returns, volatility,...

We always return to the basic issue of storing data for the underlying prices. The class implemented in the following is an attempt to solve that problem once and for all. The purpose of the `security_price_history` class is to hold a *price history* for a security. The emphasis is on efficiently storing this data. The general idea is to store the data in the following form:

Security Name

Date	bid price	trade price	ask price
2 jan 1990	99	100	101
3 jan 1990	98	99	100
.	.	.	.
.	.	.	.

One assumption made here is that the data is not on a higher frequency than daily. If so need to change the `date` class to a `date_time` class that also stores time of day.

Calculating returns etc will then be simply provided as functions that work on this data structure.

This class is also useful as a base class. For example, a stock will need added functions for dividends, adjustments etc.

5.1 Header file

The header file defines the class interface.

```
#ifndef _SECURITY_PRICE_HISTORY_H_
#define _SECURITY_PRICE_HISTORY_H_

#include "dated_obs.h"

class security_price_history {
private:
    string          security_name_;
    vector<date> dates_;
    vector<double> bids_;
    vector<double> trades_;
    vector<double> asks_;
public:
    security_price_history();
    security_price_history(const string&);
    security_price_history(const security_price_history&);
    security_price_history operator=(const security_price_history&);
    ~security_price_history();
    void clear();
    void set_security_name(const string&);
    void add_prices(const date& d, const double& bid, const double& trade, const double& ask);
    // the most important function, all dates is added through calls to add_prices

    void set_bid (const int& i, const double& bid); // to change current data.
    void set_trade (const int& i, const double& trade);
    void set_ask (const int& i, const double& ask);

    void erase(const date&); // delete on given dates
    void erase_before(const date&);
    void erase_between(const date&, const date&);
    void erase_after(const date&);
    bool contains(const date&) const; // check whether this date is present
private:
    int index_of_date(const date&) const;// where in vector is this date?
    int index_of_last_date_before(const date&) const ;
    int index_of_first_date_after(const date&) const ;
public:
    bool empty() const;
    int size() const;// { return dates_.size(); };
    string security_name() const;
    vector<date> dates() const;

    int no_dates() const;
    int no_prices() const;
    int no_bids() const;
    int no_trades() const;
    int no_asks() const;
    int no_prices_between(const date&, const date&) const;
    date date_at(const int&) const;
    double bid(const int&) const;
    double bid(const date&) const;
    double trade(const int&) const;
    double trade(const date&) const;
    double ask(const int&) const;
    double ask(const date&) const;
    date first_date() const;
    date last_date() const;
    double price(const date&) const; // price at given data
    double current_price(const date&) const; // price at or before given data
    double price(const int&) const;
    double buy_price(const int&) const;
    double sell_price(const int&) const;
};

dated_observations prices(const security_price_history&);
vector<date> dates(const security_price_history&);

#endif
```


Header file 5.1: security price history.h

5.2 Implementation

```
#include "security_price_history.h"

security_price_history:: security_price_history(){ clear(); }; // make sure empty

security_price_history:: security_price_history(const string& name){
   clear();
   security_name_ = name;
};
security_price_history:: ~security_price_history(){ clear(); };

// copy construction
security_price_history:: security_price_history(const security_price_history& sh) {
    clear();
    dates_ = vector<date>(sh.size());
    bids_  = vector<double>(sh.size());
    trades_ = vector<double>(sh.size());
    asks_  = vector<double>(sh.size());
    for (unsigned i=0;i<sh.no_dates();++i){
        dates_[i]=sh.date_at(i);
        bids_[i]=sh.bid(i);
        trades_[i]=sh.trade(i);
        asks_[i]=sh.ask(i);
    };
    set_security_name(sh.security_name());
};

security_price_history security_price_history:: operator= (const security_price_history& sh) {
    clear();
    dates_ = vector<date>(sh.size());
    bids_ = vector<double>(sh.size());
    trades_= vector<double>(sh.size());
    asks_ = vector<double>(sh.size());
    for (unsigned i=0;i<sh.no_dates();++i){
        dates_[i]=sh.date_at(i);
        bids_[i]=sh.bid(i);
        trades_[i]=sh.trade(i);
        asks_[i]=sh.ask(i);
    };
    set_security_name(sh.security_name());
    return (*this);
};

void security_price_history::clear() {
    dates_.erase(dates_.begin(),dates_.end());
    bids_.erase(bids_.begin(),bids_.end());
    trades_.erase(trades_.begin(),trades_.end());
    asks_.erase(asks_.begin(),asks_.end());
    security_name_=string();
};
string security_price_history::security_name() const { return security_name_; };
void security_price_history::set_security_name(const string& s)      { security_name_ = s; };
date  security_price_history::date_at(const int& t) const { return dates_[t]; };

bool security_price_history::empty() const{
    if ( (no_dates()<1) && (security_name().length()<1) ) return true;
    return false;
};

void security_price_history::set_bid (const int& i, const double& bid) { if (i<no_dates()) bids_[i]=bid; };
void security_price_history::set_trade (const int& i, const double& trade) { if (i<no_dates()) trades_[i]=trade; };
void security_price_history::set_ask (const int& i, const double& ask) { if (i<no_dates()) asks_[i]=ask; };

double security_price_history::bid (const int& t) const { return bids_[t]; };
double security_price_history::trade(const int& t) const { return trades_[t]; };
double security_price_history::ask (const int& t) const { return asks_[t]; };
```


C++ Code 5.1: Implementing the class

```
#include "security_price_history.h"

int security_price_history::no_dates () const {
    return int(dates_.size());
};

int security_price_history::no_prices() const {
    int n_prices=0;
    for (int i=0;i<no_dates();++i){
        if ( (bid(i)>0) || (trade(i)>0) || (ask(i)>0) ) { ++n_prices; };
    };
    return n_prices;
};

int security_price_history::no_prices_between(const date& first, const date& last) const {
    int n_pric_between=0;
    for (int i=0;i<no_dates();++i) {
        if  ((date_at(i)>=first) && (date_at(i)<=last)) {
            if((bid(i)>0)||(trade(i)>0)||(ask(i)>0)) {++n_pric_between;};
        };
    };
    return n_pric_between;
};

int security_price_history::no_bids() const {
    int n_prices=0;
    for (int i=0;i<no_dates();++i){
        if (bid(i)>0) { ++n_prices; };
    };
    return n_prices;
};

int security_price_history::no_trades() const {
    int n_prices=0;
    for (int i=0;i<no_dates();++i){
        if (trade(i)>0) { ++n_prices; };
    };
    return n_prices;
};

int security_price_history::no_asks() const {
    int n_prices=0;
    for (int i=0;i<no_dates();++i){
        if (ask(i)>0) { ++n_prices; };
    };
    return n_prices;
};
```

C++ Code 5.2: Sizes

```
#include "security_price_history.h"
const double MISSING_OBS=-1;
double security_price_history::price(const int& i) const{
    if (i>=no_dates()) { return MISSING_OBS; };
    if (trade(i)>0.0) { return trade(i); };
    if ( (bid(i)>0.0) && (ask(i)>0.0)) { return 0.5*(bid(i)+ask(i)); }; // return bid ask average
    if (bid(i)>0.0)  { return bid(i); }; // to avoid big jumps,
    if (ask(i)>0.0)  { return ask(i); }; // skip these
    return MISSING_OBS;
};

double security_price_history::bid(const date& d) const {
  int i = index_of_date(d); // return price on date, only if obs on that date, else return missing
  if (i>=0) return bid(i);
  return MISSING_OBS;
};

double security_price_history::trade(const date& d) const {
  int i = index_of_date(d);
  if (i>=0) return trade(i);
  return MISSING_OBS;
};

double security_price_history::ask(const date& d) const {
  int i = index_of_date(d);
  if (i>=0) return ask(i);
  return MISSING_OBS;
};

double security_price_history::price(const date& d) const {
  int i = index_of_date(d);
  if (i>=0) return price(i);
  return MISSING_OBS;
};

double security_price_history::current_price(const date& d) const { // return price on given date.
    if (empty())        { return MISSING_OBS; };
    if (d<first_date()) { return MISSING_OBS; }; // if before first or after last, return missing
    if (d>last_date()) { return MISSING_OBS; };
    int i = index_of_date(d); // If don't have that price, return the last one observed before the wanted date
    if (i>=0) return price(i);
    i=index_of_last_date_before(d);   // otherwise use last previously observed price.
    if (i>=0) return price(i);
    return MISSING_OBS;
};

double security_price_history::buy_price(const int& i) const{
    if (i>=no_dates()) { return MISSING_OBS; };
    if (ask(i)>0.0)  { return ask(i); };
    if (trade(i)>0.0) { return trade(i); };
    return MISSING_OBS;
};

double security_price_history::sell_price(const int& i) const{
    if (i>=no_dates()) { return MISSING_OBS; };
    if (bid(i)>0.0)  { return bid(i); };
    if (trade(i)>0.0) { return trade(i); };
    return MISSING_OBS;
};

dated_observations prices(const security_price_history& sh ){
    dated_observations pr;
    for (int i=0;i<sh.no_prices();++i) {
        date d=sh.date_at(i);
        double price = sh.price(i);
        if (price>0) pr.insert(d,price);
    };
    return pr;
};
```


C++ Code 5.3: Prices

```
#include "security_price_history.h"


void security_price_history::add_prices(const date& d,
                                        const double& bid,
                                        const double& trade,
                                        const double& ask) {
    if (!d.valid()) return; // don't add
    if ( (bid<0) && (trade<0) && (ask<0) ) return; // don't bother about empty dates
    if ( (no_dates()<1) || (d>last_date() ) ) {
        dates_ .push_back(d);
        trades_.push_back(trade);
        bids_ .push_back(bid);
        asks_ .push_back(ask);
        return;
    };
    if (d<first_date()) {
        dates_.insert(dates_.begin(),d);
        bids_.insert(bids_.begin(),bid);
        trades_.insert(trades_.begin(),trade);
        asks_.insert(asks_.begin(),ask);
        return;
    };
    int i = index_of_date(d);
    if (i>=0) { // found, replace
        trades_[i]=trade;
        bids_[i]=bid;
        asks_[i]=ask;
        return;
    }
    // evidently should be inserted somewhere in between other observations.
    for (i=0;i<no_dates();++i){
        if (date_at(i)>d) {
            dates_.insert(dates_.begin()+i,d);
            bids_.insert(bids_.begin()+i,bid);
            trades_.insert(trades_.begin()+i,trade);
            asks_.insert(asks_.begin()+i,ask);
            return;
        };
    };
};


int security_price_history::size() const { return int(dates_.size()); };

date security_price_history::first_date() const {
    if (dates_.size()>0) { return dates_.front();};
    return date(); // else
};

date security_price_history::last_date() const {
    if (dates_.size()>0) { return dates_.back();};
    return date();
};

vector<date> security_price_history::dates() const {
    return dates_;
};
```

C++ Code 5.4: Utilities

```
#include "security_price_history.h"

void security_price_history::erase(const date& d) {
   int i = index_of_date(d);
   //   cout << "erasing" << endl;
   if (i>=0) { // returns -1 if not there
       //       cout << "erasing" << endl;
      dates_.erase(dates_.begin()+i);
      trades_.erase(trades_.begin()+i);
      bids_.erase(bids_.begin()+i);
      asks_.erase(asks_.begin()+i);
   };
};

void security_price_history::erase_between(const date& first,
                                           const date& last) {
   int i1 = index_of_first_date_after(first);
   int i2 = index_of_last_date_before(last);
   if ( (i1>=0) && (i2>=0) ) { // returns -1 if not there
      i2++; // the erase does not delete the last one.
      dates_.erase(dates_.begin()+i1,dates_.begin()+i2);
      trades_.erase(trades_.begin()+i1,trades_.begin()+i2);
      bids_.erase(bids_.begin()+i1,bids_.begin()+i2);
      asks_.erase(asks_.begin()+i1,asks_.begin()+i2);
   };
};

void security_price_history::erase_before(const date& d) {
   int i = index_of_last_date_before(d);
   if (i>=0) { // returns -1 if not there
      i++;// the erase does not delete the last one, so add one.
      dates_.erase(dates_.begin(),dates_.begin()+i);
      trades_.erase(trades_.begin(),trades_.begin()+i);
      bids_.erase(bids_.begin(),bids_.begin()+i);
      asks_.erase(asks_.begin(),asks_.begin()+i);
   };
};

void security_price_history::erase_after(const date& d) {
   int i = index_of_first_date_after(d);
   if (i>=0) { // returns -1 if not there
      dates_.erase(dates_.begin()+i,dates_.end());
      trades_.erase(trades_.begin()+i,trades_.end());
      bids_.erase(bids_.begin()+i,bids_.end());
      asks_.erase(asks_.begin()+i,asks_.end());
   };
};
```

C++ Code 5.5: Erasing

```
#include "security_price_history.h"

bool security_price_history::contains(const date& d) const {
    return binary_search(dates_.begin(),dates_.end(),d);
};

int security_price_history::index_of_date(const date& d) const {
   // this routine returns the index at which date d is, or -1 if not found.
   if (!contains(d)) return -1;
   for (int i=0;i<dates_.size();++i){
       if (dates_[i]==d) return i;
   };
   return -1;
};

int security_price_history::index_of_first_date_after(const date& d) const {
    if (!d.valid()) return -1;
    if (d>=last_date()) return -1;
    if (d<first_date()) return 0;
    for (int i=0;i<dates_.size();++i){
         if (dates_[i]>d) return i;
    };
    return -1;
};

int security_price_history::index_of_last_date_before(const date& d) const {
    if (!d.valid()) return -1;
    if (d<=first_date()) return -1;
    if (d>last_date()) return index_of_date(last_date());
    for (int i=0;i<dates_.size();++i){
         if (dates_[i]>=d) return i-1;
    };
    return -1;
};
```

C++ Code 5.6: Searching

Chapter 6

Stock price history

The following class is used when we have a lot of historical data about stocks. Given price observations, do a lot of various calculations and pulling of various data, as well as printing various reports.

6.1 Stock history

The purpose of the `stock_price_history` class is to hold a price history for a stock, with a number of special functions for that purpose.

The class is made by adding data and functions to the base class `security_price_history`. The reason the general security price history class is not enough, is the problem that dividends and other adjustments need to accounted for in returns calculations.

6.2 Header file

6.2.1 stock_price_history.h

Note that a lot of the functionality of the class is inherited from `security_price_history`

```
#ifndef _STOCK_PRICE_HISTORY_H_

#include "security_price_history.h"
#include "dated_obs.h"

class stock_price_history : public security_price_history { // most of the functionality is inherited from security price history
private:
    dated_observations dividends_;
    dated_observations adjustments_;
public:
    stock_price_history();
    stock_price_history(const stock_price_history&);
    stock_price_history operator = (const stock_price_history&);
    ~stock_price_history(){ clear(); };
    void clear();

    bool empty() const;
    ////////////////////// dividends //////////////////////////
    void     add_dividend (const date&, const double&);
    void     remove_dividend (const date&);
    int      no_dividends() const;
    date     dividend_date(const int&) const;
    double   dividend(const int&) const;
    bool     dividends_between(const date&, const date&) const;
    int      no_dividends_between(const date&, const date&) const;
    double   total_dividends_between(const date&, const date&) const;
    bool     dividend_on(const date&) const;
    double   dividend(const date&) const;

    ///// adjustments ////////////////////////////////
    void   add_adjustment      (const date&, const double&);
    void   remove_adjustment   (const date&);
    int    no_adjustments() const;
    date   adjustment_date(const int& i) const;
    double adjustment(const int& i) const;
    bool   adjustments_between(const date& d1, const date& d2) const;
    int    no_adjustments_between(const date& d1, const date& d2) const;
    double aggregated_adjustments_between(const date& d1, const date& d2) const;
    bool   adjustment_on(const date& d) const;
    double adjustment(const date& d) const;
};

dated_observations daily_prices    (const stock_price_history& st);
dated_observations daily_prices    (const stock_price_history& st, const date& from, const date& to);
dated_observations daily_trade_prices (const stock_price_history& st);
dated_observations daily_trade_prices (const stock_price_history& st, const date& from, const date& to);
dated_observations monthly_prices (const stock_price_history& st);
dated_observations annual_prices  (const stock_price_history& st);

dated_observations dividends(const stock_price_history&);
dated_observations dividends(const stock_price_history&, const date& d1, const date& d2);

dated_observations adjustments(const stock_price_history&);
dated_observations adjustments(const stock_price_history&, const date& d1, const date& d2);

#define _STOCK_PRICE_HISTORY_H_
#endif
```

Header file 6.1: Define all class elements

6.3 Implementation

```
#include "stock_price_history.h"

const double MISSING_OBS = −1; // Assumption: Prices are positive. Negative prices are missing observations.

stock_price_history::stock_price_history(){; };

stock_price_history::stock_price_history(const stock_price_history& sh): security_price_history(sh){
   dividends_.clear();
   for (unsigned i=0;i<sh.no_dividends();++i)
      add_dividend(sh.dividend_date(i), sh.dividend(i));
   adjustments_.clear();
   for (unsigned i=0;i<sh.no_adjustments();++i)
      add_adjustment(sh.adjustment_date(i), sh.adjustment(i));
};

stock_price_history stock_price_history:: operator= (const stock_price_history& sh) {
   clear();
   security_price_history::operator=(sh);
   for (unsigned i=0;i<sh.no_dividends();++i)
      add_dividend(sh.dividend_date(i), sh.dividend(i));
   for (unsigned i=0;i<sh.no_adjustments();++i)
      add_adjustment(sh.adjustment_date(i), sh.adjustment(i));
   return *this;
};

void stock_price_history::clear() {
    security_price_history::clear();
    dividends_.clear();
    adjustments_.clear();
};

bool stock_price_history::empty() const {
   if (no_dividends()>0) return false;
   if (no_adjustments()>0) return false;
   return security_price_history::empty();
};
```

C++ Code 6.1: Basic operations

```
#include "stock_price_history.h"
// querying
int stock_price_history::no_dividends() const { return dividends_.size(); };
date stock_price_history::dividend_date(const int& i) const { return dividends_.date_at(i); };
double stock_price_history::dividend(const int& i) const { return dividends_.element_at(i); };

bool stock_price_history::dividend_on(const date& d) const { return dividends_.contains(d); };

double stock_price_history::dividend(const date& d) const {
   double div = dividends_.element_at(d);
   if (div>0) return div;
   return 0.0;
};

void stock_price_history::add_dividend(const date& dividend_date, const double& dividend_amount) {
   dividends_.insert(dividend_date,dividend_amount);
};

void stock_price_history::remove_dividend(const date& dividend_date) { dividends_.remove(dividend_date);};

bool stock_price_history::dividends_between(const date& d1, const date& d2) const {
    date first, last;
    if (d1<d2) { first=d1; last=d2; } else { first=d2; last=d1; };
    if( dividends_.no_obs_between(first,last)>0) return true;
    return false;
};

int stock_price_history::no_dividends_between(const date& d1, const date& d2) const {
    date first, last;
    if (d1<d2) { first=d1; last=d2; } else { first=d2; last=d1; };
    return dividends_.no_obs_between(first,last);
};

double stock_price_history::total_dividends_between(const date& d1, const date& d2) const {
   if (!dividends_between(d1,d2)) return 0;
   date first, last;
   if (d1<d2) { first=d1; last=d2; } else { first=d2; last=d1; };
   double tot_dividend = 0.0;
   for (int i=0; i<dividends_.size(); ++i){
      if ( (dividends_.date_at(i)>first) && (dividends_.date_at(i)<=last) ) {
          tot_dividend += dividends_.element_at(i);
      };
   };
   return tot_dividend;
};

dated_observations dividends(const stock_price_history& sphist, const date& d1, const date& d2){
    dated_observations dividends;
    for (int i=0;i<sphist.no_dividends();++i){
        date d=sphist.dividend_date(i);
        if ( (d>=d1) && (d<=d2) && (sphist.dividend(i)>=0.0) ){
            dividends.insert(d,sphist.dividend(i));
        };
    };
    return dividends;
};

dated_observations dividends(const stock_price_history& sphist){
    return dividends(sphist,sphist.first_date(),sphist.last_date());
};
```

C++ Code 6.2: Dividends

```
#include "stock_price_history.h"

int stock_price_history::no_adjustments() const{ return adjustments_.size(); };
date stock_price_history::adjustment_date(const int& i) const{ return adjustments_.date_at(i); };
double stock_price_history::adjustment(const int& i) const { return adjustments_.element_at(i);};
bool stock_price_history::adjustment_on(const date& d) const {
   for (int i=0; i<adjustments_.size(); ++i) {
      if (adjustments_.date_at(i)==d) return true;
   };
   return false;
};
double stock_price_history::adjustment(const date& d) const {
    for (int i=0; i<adjustments_.size(); ++i){ if (adjustments_.date_at(i)==d) return adjustments_.element_at(i); };
    return 0.0;
};

void stock_price_history::add_adjustment ( const date& adjustment_date, const double& adj_factor) {
   if (adjustments_.contains(adjustment_date)) {
      adjustments_.insert(adjustment_date, adjustments_.element_at(adjustment_date)*adj_factor);
   }
   else { adjustments_.insert(adjustment_date,adj_factor); };
};

void stock_price_history::remove_adjustment(const date& adjustment_date){ adjustments_.remove(adjustment_date); };

bool stock_price_history::adjustments_between(const date& d1, const date& d2) const{
    date first, last;
    if (d1<d2) { first=d1; last=d2; } else { first=d2; last=d1; };
    if (adjustments_.no_obs_between(first,last)>0) return true;
    return false;
};

int stock_price_history::no_adjustments_between(const date& d1, const date& d2) const {
    date first, last;
    if (d1<d2) { first=d1; last=d2; } else { first=d2; last=d1; };
    return adjustments_.no_obs_between(first,last);
};

double stock_price_history::aggregated_adjustments_between(const date& d1, const date& d2) const {
    date first, last;
    if (d1<d2) { first=d1; last=d2; } else { first=d2; last=d1; };
    double tot_adjustments=1.0;
    for (int i=0; i<adjustments_.size(); ++i){
        if ( (adjustments_.date_at(i)>first) && (adjustments_.date_at(i)<=last) ) {
           tot_adjustments *= adjustments_.element_at(i);
        };
    };
    return tot_adjustments;
};

dated_observations adjustments(const stock_price_history& sphist, const date& d1, const date& d2){
    dated_observations adjustments;
    for (int i=0;i<sphist.no_adjustments();++i){
        date d=sphist.adjustment_date(i);
        if ( (d>=d1) && (d<=d2) && (sphist.adjustment(i)>0) ){
            adjustments.insert(d,sphist.adjustment(i));
        };
    };
    return adjustments;
};

dated_observations adjustments(const stock_price_history& sphist){
    return adjustments(sphist,sphist.first_date(),sphist.last_date());
};
```

C++ Code 6.3: Adjustments