```
//  Additive pattern for numeric classes
//  ------------------------------------
//                      copyright 2001 Information Disciplines, Inc., Chicago

//  These preprocessor components define the pattern of arithmetic
//  operators for a numeric data item class that follows the additive pattern.

//  Before #include-ing this file, the programmer must set the macro
//  variable  Class  to the name of the class, and, optionally, set
//  PureType to the type of the pure number (default = double).  It is up to
//  the class definition (.hpp) file to #undef these macros.

#ifndef PureType
  #define PureType double
#endif

//  The following functions must be defined in the class implementation (.cpp)
//  file or the class definition (.hpp) file.

 Class     operator- () const;
 Class&    operator+=(const Class    rs);
 Class&    operator-=(const Class    rs);
 Class&    operator*=(const PureType rs);
 Class&    operator/=(const PureType rs);
 Class&    operator%=(const PureType rs);   // Omit these definitions if
 Class&    operator%=(const Class    rs);   //   % operators are unused
PureType   operator/ (const Class    rs) const;

//  The following inline functions are fully defined here:

 Class     operator+(const Class    rs) const {return Class(*this) += rs;}
 Class     operator-(const Class    rs) const {return Class(*this) -= rs;}
 Class     operator*(const PureType rs) const {return Class(*this) *= rs;}
 Class     operator/(const PureType rs) const {return Class(*this) /= rs;}
 Class     operator%(const PureType rs) const {return Class(*this) %= rs;}
 Class     operator%(const Class    rs) const {return Class(*this) %= rs;}

 inline friend Class operator*(const PureType ls, const Class rs)
                 {return rs * ls;}
```

```
//Angle class   --   copyright 2001, Information Disciplines, Inc.
//  This class supports operations on plane angles.

#ifndef ANGLE
#define ANGLE const Angle
#include <math.h>             /*  For trig functions                     */
#include "global.hpp"         /*  For const types & template functions   */


class Angle{
  double  value;                 //  normalized radians
  static const double PI;

//  For compatibility with the standard C-library and other software, range is
//  (-pi,pi] not [0,2pi), enforced by the following function:

  void normalize();


//  Constructors
//  ------------

public:
  Angle(DOUBLE rad = 0.0)  : value(rad) {normalize();}
  Angle(INT deg, INT min, INT sec);

//  The compiler will supply an acceptable copy constructor, destructor
//  and assignment operator.


//  Accessors
//  ---------

 double toDegrees() const {return  180.0 * (value / PI);}
 double toRadians() const {return  value;}
 short  degrees()   const {return  short(toDegrees());}
 short  minutes()   const {return  long (abs(toDegrees()) * 60.0)   % 60;}
 short  seconds()   const {return  long (abs(toDegrees()) * 3600.0) % 60;}
```

```
//  Operators follow the standard additive pattern
//  ---------

#define   Class Angle
#include "Additive.hpp"

//  Trigonometric functions (for notational consistency and to hide internal
//  ----------------------  representation.  User can use other old functions
//                          by extracting value with  toRadians() accessor)

  double cos()          const {return ::cos (value);}
  double sin()          const {return ::sin (value);}
  double tan()          const {return ::tan (value);}

};                //  *************** End of class definition


inline Angle  Angle::operator- ()         const {return Angle(-value);}

inline Angle& Angle::operator+=(const Class rs)
                    {value+=rs.value; normalize(); return *this;}
inline Angle& Angle::operator-=(const Class rs)
                    {value-=rs.value; normalize(); return *this;}
inline Angle& Angle::operator*=(const PureType rs)
                    {value*=rs;       normalize(); return *this;}
inline Angle& Angle::operator/=(const PureType rs)
                    {value/=rs;       normalize(); return *this;}

inline double Angle::operator/ (ANGLE  rs) const {return value / rs.value;}
```

```
//  Non-member operators
//  --------------------
    ostream& operator<< (ostream& ls, ANGLE rs);

//  WARNING:  Floating point equality test is undependable

 inline bool   operator==(ANGLE  ls, ANGLE rs)
                              {return ls.toRadians() == rs.toRadians();}

//  NOTE:  Ordering is ambiguous and not transitive, due to normalization.
 inline bool   operator<  (ANGLE  ls, ANGLE rs)
                              {return ls.toRadians() < rs.toRadians();}

//  Inverse trigonometric functions (forward to C library versions)
//  -------------------------------

 inline Angle  arccos(DOUBLE x)         {return ::acos(x);}
 inline Angle  arcsin(DOUBLE x)         {return ::asin(x);}
 inline Angle  arctan(DOUBLE y,
                      DOUBLE x = 1.0)   {return ::atan2(y,x);}
#endif
```

```
// Angle class implementation--copyright 2001 Information Disciplines, Inc.
//  (See Angle.hpp for detailed explanations)

#include "Angle.hpp"

  DOUBLE Angle::PI = 3.141592653589793;

void Angle::normalize()
    {static DOUBLE twoPi = PI + PI;
     while (value <= -PI) value += twoPi;
     while (value >   PI) value -= twoPi;
    }

  Angle::Angle(INT deg, INT min, INT sec)
    {double seconds = sec + 60 * (min + 60 * abs(deg));
     double degrees = seconds / 3600.0;
            value = (degrees * PI / 180.0) * sign(deg);
            normalize();
     }


 ostream&  operator<< (ostream& ls, ANGLE rs)
           {CHAR degreeSymbol = '\370';
            return ls << rs.degrees() << degreeSymbol
                      << rs.minutes() << '\''
                      << rs.seconds() << '\"';}
```

```
//  Calendar Information (copyright 1992, Information Disciplines, Inc.)
//  --------------------

//  This pseudo-class defines public constants and utility functions
//  used in manipulating dates.  It is independent of any date representation
//  or Date class.  (It is used by IDI's Date class -- see Date.hpp.)

#ifndef CALENDAR
#define CALENDAR

#include "SimpleString.hpp"

namespace CalendarInfo {

//  Constants
//  ---------

  static SHORT   DAYS_PER_YEAR      =  365;
  static LONG    DAYS_PER_4_YEARS   =  1 +  4 * DAYS_PER_YEAR;
  static LONG    DAYS_PER_100_YEARS = -1 + 25 * DAYS_PER_4_YEARS;
  static LONG    DAYS_PER_400_YEARS =  1 +  4 * DAYS_PER_100_YEARS;

  static SHORT   DAYS_IN_MONTH     [13] = {0, 31, 28, 31, 30, 31, 30,
                                           31, 31, 30, 31, 30, 31};
  static SHORT   DAYS_BEFORE_MONTH [13] = {0,  0, 31, 59, 90,120,151,
                                          181,212,243,273,304,334};

  static const char *const  MONTH_NAME        [13]
    = {"","January", "February", "March"    , "April"  , "May"     , "June" ,
       "July"   , "August"  , "September", "October", "November", "December"};

  static const char *const  DAY_NAME          [ 7] =
     {"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"};
```

```
//  Utility function declarations (defined in CalendarInfo.cpp)
//  ----------------------------

   bool isLeapYear(INT yyyy);

   bool isLegalYMD(INT yyyy, SHORT mm, SHORT dd);

   short dayNumber(INT yyyy, INT mm, INT dd);

   void ymd       (INT yyyy_in, short& mm_out, short& ddd_in_out); // Inverse of
                                                                   //   dayNumber
   SimpleString toEnglish(INT yyyy, SHORT mm, SHORT dd);

   SimpleString toString(INT yyyy, SHORT mm, SHORT dd);
 };

#endif
```

```
// CalendarInfo implementation (1992, Information Disciplines, Inc.)
// -------------------------   (See CalendarInfo.hpp for details.)

#include "global.hpp"
#include "CalendarInfo.hpp"       /*  Declarations, also included by users */

//  Calendar arithmetic functions
//  -----------------------------

  bool CalendarInfo::isLeapYear (INT yyyy)
  {return (0 == yyyy % 400) || ((0 == yyyy % 4)  && (0 != yyyy % 100));}

  bool CalendarInfo::isLegalYMD (INT yyyy, SHORT mm, SHORT dd)
  {return mm > 0 &&  mm <= 12
     &&   dd > 0 && (dd <= DAYS_IN_MONTH[mm]
                       || (dd == 29 && mm == 2 && isLeapYear(yyyy)));
  }

  short CalendarInfo::dayNumber                    //  Convert year-month-day
           (INT yyyy, INT mm, INT dd)              //    to year and day number
  {if  (!isLegalYMD(yyyy, mm, dd)) return 0;       //  (Sometimes mistakenly
   int  ddd = DAYS_BEFORE_MONTH[mm] + dd;          //    called a "Julian date")
   if  (isLeapYear(yyyy) && mm > 2) ++ddd;
   return ddd;
  }
```

```
//  This routine performs the inverse of the above dayNumber function.
//  (No error checking -- result undefined for day number out of range)

  void CalendarInfo::ymd  (INT yyyy_in, short& mm_out, short& ddd_in_out)
  {ddd_in_out %= 366;
   if (isLeapYear(yyyy_in))              //  For leap years
     if  (ddd_in_out > 60) --ddd_in_out; //     adjust day after 29 Feb.,
     else if (ddd_in_out == 60)          //  Test for 29 Feb. special case
          {mm_out = 2; ddd_in_out = 29; return;}

   mm_out = (ddd_in_out  + 28) / 29;     //  Estimate the month, then adjust
   if  (mm_out >= 13
        || ddd_in_out <= DAYS_BEFORE_MONTH[mm_out])
        --mm_out;

   ddd_in_out -= DAYS_BEFORE_MONTH[mm_out];//  Compute day of month
   return;
  }

//  Conversion functions to external representations
//  --------------------

   SimpleString CalendarInfo::toString(INT yyyy, SHORT mm, SHORT dd)
    {return SimpleString::toString(yyyy,10) + '-'
          + SimpleString::toString(mm,10)   + '-'
          + SimpleString::toString(dd,10);
    }

   SimpleString CalendarInfo::toEnglish           //  Date to American English
            (INT yyyy, SHORT mm, SHORT dd)        //     conversion
   {return SimpleString(MONTH_NAME[mm])
           + ' '  + SimpleString::toString(dd,10)
           + ", " + SimpleString::toString(yyyy,10);
   }
```

```
//Complex class definition (copyright 1993, Information Disciplines, Inc.)
//  All functions are in-line. There's no separate implementation code file.

//  For polar coordinates, this class will use the Angle class if the user
//  includes it prior to this file.

#ifndef COMPLEX                /*  Multiple definition guard             */
#define COMPLEX const Complex  /*  Conventional notation for constants   */

#include <math.h>              /*  For sqrt (in abs) and atan (in theta) */
#include "global.hpp"


class Complex {
   double   rl, im;                  //  Real & imaginary parts

//  Constructors and destructor
//  ---------------------------
public:
   Complex(DOUBLE x = 0, DOUBLE y = 0) : rl(x), im(y) {} //  Real & imaginary
 #ifdef ANGLE
   Complex(DOUBLE r,     ANGLE  t)     : rl(r*t.cos()),  //  Rho & theta
                                         im(r*t.sin()) {}
 #endif
//  The compiler will generate by default an appropriate destructor,
//  copy constructor, and assignment operator.
```

```
//  Accessors
//  ---------
   double realPart() const {return rl;}
   double imagPart() const {return im;}
   double rho()      const                          //  Magnitude of vector
         {return sqrt(realPart() * realPart()    //    from origin of
                    + imagPart() * imagPart());} //    complex plane
  #ifdef ANGLE
    Angle
  #else
    double
  #endif
     theta()    const                               //  Angle between vector
          {return atan2(imagPart(),realPart());}  //    & real-axis


//  Member operators
//  ----------------
   Complex operator-() const        //  Unary minus operator
   {return Complex(-rl, -im);}

   operator double ()  const        //  Conversion to real
   {assert(im == 0); return rl;}    //  (Provided that value is real)

};           //   ************       End of class definition


//  Non-member operators and functions
//  ----------------------------------
//  These functions are neither member nor friend functions, since they
//  can gain access to the component data through the accessors.

 inline  double  abs (COMPLEX x)    //  Magnitude or absolute value
        {return x.rho();}

 inline ostream& operator<< (ostream& ls,    //  External representation is
                             COMPLEX  rs)    //    ordered pair in parens.
 {ls << '(' << rs.realPart() << ", " << rs.imagPart() << ')'; return ls;}
```

```
//  Arithmetic operations
//  ---------------------

 inline Complex operator+ (COMPLEX ls, COMPLEX rs)
   {return Complex(ls.realPart() + rs.realPart(),
              ls.imagPart() + rs.imagPart());}

 inline Complex operator- (COMPLEX ls, COMPLEX rs) {return ls + (-rs);}

 inline Complex operator* (COMPLEX ls, COMPLEX rs)
   {return Complex
        (ls.realPart() * rs.realPart() - ls.imagPart() * rs.imagPart(),
         ls.realPart() * rs.imagPart() + ls.imagPart() * rs.realPart());}

 inline Complex operator/ (COMPLEX ls, COMPLEX rs)
   {DOUBLE  denom = rs.realPart() * rs.realPart()
                    + rs.imagPart() * rs.imagPart();
    return   Complex ((ls.realPart() * rs.realPart()
                   +  ls.imagPart() * rs.imagPart()) / denom,
                     (rs.realPart() * ls.imagPart()
                   -  rs.imagPart() * ls.realPart()) / denom);}

 inline Complex& operator+= (Complex& ls, COMPLEX rs) {return ls = ls + rs;}
 inline Complex& operator-= (Complex& ls, COMPLEX rs) {return ls = ls - rs;}
 inline Complex& operator*= (Complex& ls, COMPLEX rs) {return ls = ls * rs;}
 inline Complex& operator/= (Complex& ls, COMPLEX rs) {return ls = ls / rs;}


//  Relational operator
//  ----------
 inline bool   operator== (COMPLEX ls, COMPLEX rs)
      {return  ls.realPart() == rs.realPart()
           &&  ls.imagPart() == rs.imagPart();}
#endif
```

```
// Duration Days and Date classes (Copyright 1994,
// ----------------------------        Information Disciplines, Inc.)

//  These two classes support Date objects as well as duration objects
//  measured in Days.  They follow the Point-Extent pattern.  Dependencies
//  on the Gregorian calendar are minimized through the use of the
//  CalendarInfo pseudo-class.

#ifndef DAYS
#define DAYS const Days
#define DATE const Date

//   I.  Duration (extent) class in number of days
//   --------------------------

class Days {
    long  value;              //  Internal representation # of days

friend class Date;            //  Allow Date methods to retrieve value


//  Constructors
//  ------------

public:
    Days(LONG x = 0) : value(x) {}

//    Compiler will supply acceptable copy constructor, destructor,
//    and assignment operator.  There are no accessors.


//  Arithmetic Operators
//  --------------------

#define Class Days
#define PureType long
#include "Additive.hpp"     /*  Additive pattern for binary operators */

DAYS& operator++()      {++value; return *this;}
DAYS& operator--()      {--value; return *this;}
DAYS  operator++(int)   {DAYS result = *this; ++value; return result;}
DAYS  operator--(int)   {DAYS result = *this; --value; return result;}
```

```
//  Relational operators
//  --------------------

 bool operator== (DAYS rs) const {return value == rs.value;}
 bool operator<  (DAYS rs) const {return value <  rs.value;}

//  The remaining relational operators will be instantiated from the
//  template in global.hpp.


//  I-O operations
//  --------------

 static char*  unit;        //  Unit name ("day") for messages
 ostream& put(ostream& s) const;
 istream& get(istream& s);

};               //  ************   End of class definition

//  Inline member and non-member functions
//  --------------------------------------

inline ostream& operator<< (ostream& ls, DAYS rs) {return rs.put(ls);}
inline istream& operator>> (istream& ls, Days& rs){return rs.get(ls);}

inline Days& Days::operator+=(DAYS rs) {value += rs.value; return *this; }
inline Days& Days::operator-=(DAYS rs) {value -= rs.value; return *this; }
inline Days& Days::operator*=(LONG rs) {value *= rs;       return *this; }
inline Days& Days::operator/=(LONG rs) {value /= rs;       return *this; }
inline long  Days::operator/ (DAYS rs) const {  return value / rs.value; }
inline Days& Days::operator%=(DAYS rs) {value %= rs.value; return *this; }
inline Days& Days::operator%=(LONG rs) {value %= rs;       return *this; }
inline Days  Days::operator- ()        const {return Days(-value);       }

//  Non-inline functions are defined in Date.cpp
```

```
//   II.  Date class
//   ---------------

//  This class defines date objects.  Although it assumes the
//  Gregorian calendar, dependencies on the calendar are minimized
//  through the CalendarInfo pseudo-class.

#include "CalendarInfo.hpp"       /*  Tables, constants, and calendar functions */


class Date  {
    long  value;              //  Internal representation:  # of days since

//  The value is the number of days since the origin defined by:

 static LONG  BIAS;           //  (See Date.cpp for value)
 static SHORT BIAS_WEEKDAY;   //  Day of week of origin date

//  This representation is compatible with some database and spreadsheet
//  software products.  Note that:

//    1.  Some conversion functions assume the Gregorian calendar,
//        even for dates before that calendar was adopted.

//    2.  B.C. dates can be generated by arithmetic operations, but
//        may not be supported by conversion functions.


 static long       yyyy;                    //  Cache storage for result of
 static short      mm;                      //    component date fields
 static short      dd;                      //   (see set_ymd() function)
 static short      ddd;
 static long       cur_value;               //  Date corresponding to
                                            //    above components
```

```
//  Constructors and accessors
//  --------------------------

public:
 Date() {}                            // (No default value)
 Date(LONG yyyy, UINT ddd);           //  Year and day number
 Date(LONG yyyy, SHORT mm, SHORT dd); //  Year-month-day
 Date(CHAR  YYMMDD[6]);               //  ANSI 6-character repr.

 static short century_break;            //  Break point for
                                        //    2-digit year

//   Compiler will supply acceptable copy constructor, destructor,
//   and assignment operator

private:  void set_ymd() const;                //  Extract date subfields
public:

 long  year()    const {set_ymd(); return yyyy;}
 short month()   const {set_ymd(); return mm;}
 short day()     const {set_ymd(); return dd;}
 short dayno()   const {set_ymd(); return ddd;}
 short weekday() const {return (((value + BIAS_WEEKDAY) % 7) + 7) % 7;}

//  Arithmetic Operators

#define PointClass Date                 /*  Point-extent pattern for combinations       */
  #include "PointExt.hpp"               //    of Date and Days objects                  */

DATE& operator++()      {++value; return *this;}
DATE& operator--()      {--value; return *this;}
DATE  operator++(int)   {DATE result = *this; ++value; return result;}
DATE  operator--(int)   {DATE result = *this; --value; return result;}


//  Relational operators
//  --------------------

 bool operator== (DATE rs) const {return value == rs.value;}
 bool operator<  (DATE rs) const {return value <  rs.value;}

//  The remaining relational operators will be instantiated from the
//  template in global.hpp
```

```
//  I-O operations

//  I-O functions
//  -------------

ostream& put(ostream& s) const;
istream& get(istream& s);

};          //  **************  End of class definition


//  Inline member and non-member functions

inline ostream& operator<< (ostream& ls, DATE rs) {return rs.put(ls);}
inline istream& operator>> (istream& ls, Date& rs){return rs.get(ls);}

inline Date& Date::operator+= (DAYS rs) {value += rs.value; return *this;}
inline Date& Date::operator-= (DAYS rs) {value -= rs.value; return *this;}

//  Non-inline functions are defined in Date.cpp

#undef Class
#undef PointClass
#endif
```

```
// Days and Date Date Implementation
// -----------------  See IDIDATE.HPP for documentation

#include "global.hpp"
#include "Date.hpp"
#include <string.h>

//  Days implementation
//  -------------------


 ostream& Days::put(ostream& ls) const
 {ls << value << ' ' << Days::unit;
  if (abs(value) != 1) ls << 's';    //  Append English plural
  return ls;
 }

 istream& Days::get(istream& ls)
   {ls >> value; return ls; }        //  (mainly for debugging)



//  Constant initialization
//  -----------------------

 LONG Date::BIAS                                 // Internal representation
      = - CalendarInfo::DAYS_PER_400_YEARS * 5  // Origin date is December 30,
        + CalendarInfo::DAYS_PER_100_YEARS      //   1899, for compatibility
        + CalendarInfo::DAYS_PER_YEAR + 2;      //      with Lotus 1-2-3
SHORT Date::BIAS_WEEKDAY = 6;                   // Origin date is a Saturday

char*   Days::unit  = "day";

short Date::century_break = 20;           // 2-digit year is 20yy if yy less
                                          //                 19yy otherwise
                                          // (user may override this one)
```

```
Date::Date (LONG yyyy, UINT ddd)          //  Year and day number
             : value(BIAS)                //  (no error check on ddd)
 {LONG years = yyyy - 1;                  //  Number of elapsed years
   value += years                         //  Convert years
          * CalendarInfo::DAYS_PER_YEAR   //    to days
          + years / 4                     //  Apply
        - years / 100                     //    leap-year
        + years / 400                     //      adjustment
          + ddd;                          //  Add day number within year
  }



Date::Date (LONG yyyy, SHORT mm, SHORT dd)    //  Year-month-day
      : value(0)                          //  (with full error checking)
  {bool leap_year = CalendarInfo::isLeapYear(yyyy);
   if   (mm < 1 ||  mm > 12
                || dd < 1 || (dd > CalendarInfo::DAYS_IN_MONTH [mm]
                      && !(dd == 29 && mm == 2 && leap_year)))
      {clog << endl << "DATE01 -- Illegal value ("
            << yyyy << '-' << mm << '-' << dd << "). ";}
      {int ddd = CalendarInfo::DAYS_BEFORE_MONTH [mm] + dd;
             ddd += (leap_year && (mm > 2)); //  First convert mm-dd to ddd
         value = Date(yyyy,ddd).value;}     //  Now use previous constructor
  }
```

```
Date::Date (CHAR yymmdd[6])                  //  ANSI 6-character string
  {if (strlen(yymmdd) != 6) return;          //  (check only on length)
   char charVal[6];
   for (int i = 0; i < 6; i++)               //  Convert characters
     charVal[i] = yymmdd[i] - '0';           //    to their values

   int   yy = charVal[0] * 10 + charVal[1];  //  Extract year
   short mm = charVal[2] * 10 + charVal[3];  //  Extract month
   short dd = charVal[4] * 10 + charVal[5];  //  Extract day

   yy += (yy < century_break) ? 2000 : 1900; //  Choose 20th or 21st century

   value = Date(yy,mm,dd).value;            //  Now use previous constructor
  }


ostream& Date::put(ostream& rs) const
 {return rs << year() << '-' << month()
                      << '-' << day();}

istream& Date::get(istream& rs)
 {int y, m, d;
  rs >> y >> m >> d;
  value = Date(y,m,d).value;
  return rs;
 }
```

```
long        Date::yyyy = 0;              // Meaningless
short       Date::mm   = 0;              //   initializations
short       Date::dd   = 0;              //     required by the
short       Date::ddd  = 0;              //       linking loader
long        Date::cur_value = 0;


void Date::set_ymd() const         //  Extract date subfields
{if  (value == cur_value) return;  //  To avoid redundant calculation we
 cur_value  =  value;              //     save the results, but this won't
                                   //     won't work with multi-tasking.
 long          ndays = value - BIAS;
 int           ngrps;

 ngrps      =  ndays / CalendarInfo::DAYS_PER_400_YEARS;
 yyyy       =  ngrps * 400;
 ndays     -=  ngrps * CalendarInfo::DAYS_PER_400_YEARS;

 ngrps      =  ndays / CalendarInfo::DAYS_PER_100_YEARS;
 yyyy      +=  ngrps * 100;
 ndays     -=  ngrps * CalendarInfo::DAYS_PER_100_YEARS;

 ngrps      =  ndays / CalendarInfo::DAYS_PER_4_YEARS;
 yyyy      +=  ngrps * 4;
 ndays     -=  ngrps * CalendarInfo::DAYS_PER_4_YEARS;

 yyyy      +=  ndays / CalendarInfo::DAYS_PER_YEAR + 1;
 ndays     %=  CalendarInfo::DAYS_PER_YEAR;

 if   (ndays != 0)                                 // Test for year end
       ddd  = short(ndays);                        // No -- set days
 else  ddd = CalendarInfo::DAYS_PER_YEAR           // Yes-- adjust to end
           + CalendarInfo::isLeapYear(-- yyyy);    //       of prev. year

//  At this point, ddd is the day number within yyyy

  CalendarInfo::ymd(yyyy, mm, ddd); dd = ddd;      // Convert to month and day

  return;}
```

```
//Global definitions to be used by any part of any program

#ifndef INT

//  #include this file at the start of each compilable (.cpp) file.
//  Then other #include files can assume that these definitions
//  are already made and need not #include this file themselves.

//  I.  Standard C and C++ Library Definitions
//  ------------------------------------------
    #include <iostream.h>   //  C++ stream I-O
    #include <assert.h>     //  Debugging (assertion) macro


//  II.  Macro Definitions
//  ----------------------
//  To save horizontal space in declarations, improve
//  program readability, and encourage use of "const"

    #define INT     const    int
    #define SHORT   const    short
    #define LONG    const    long
    #define CHAR    const    char
    #define BOOL    const    bool
    #define FLOAT   const    float
    #define DOUBLE  const    double

    #define uint    unsigned int
    #define ushort  unsigned short
    #define ulong   unsigned long
    #define uchar   unsigned char

    #define UINT    const    uint
    #define USHORT  const    ushort
    #define ULONG   const    ulong
    #define UCHAR   const    uchar
```

```
//  Global definitions (continued)


//  III. Generic Functions
//  ----------------------

#define tpl1 template<class T>            inline      /* Local macros to    */
#define tpl2 template<class T1, class T2> inline      /*  reduce repetition */


//    A.  Utility and simple arithmetic
//      ----------------------------
tpl1  T     abs  (T X)          {return X > 0      ?  X  : -X; }
tpl2  T1    min  (T1 X, T2 Y)   {return X < (T1) Y ?  X  :  Y; }
tpl2  T1    max  (T1 X, T2 Y)   {return X > (T1) Y ?  X  :  Y; }
tpl1 short  sign (T X)          {return X < 0      ? -1  :  1;}
tpl2 void   swap (T1& X, T2& Y) {T1 tempo=X; X=Y; Y=tempo; return;}


//   B.  Derived relational operators, so that classes need
//       override only  == and < as primitives.

tpl2  bool  operator!= (T1 ls, T2 rs) {return ! (ls == rs);}
tpl2  bool  operator>  (T1 ls, T2 rs) {return   (rs  < ls);}
tpl2  bool  operator<= (T1 ls, T2 rs) {return ! (ls  > rs);}
tpl2  bool  operator>= (T1 ls, T2 rs) {return   (rs <= ls);}

#undef tpl1
#undef tpl2

#endif
```

```
//  Money Class (copyright 1994, Information Disciplines, Inc)
//  -----------

//  Objects of this class are amounts of money in a standard currency
//  (default = U.S.)

#ifndef MONEY
 #define MONEY const Money
 #include "global.hpp"
 #include <math.h>

class   Money {

//   Internal representation:   An integer, scaled so that unity is
//   -----------------------      the smallest measurable quantity

    double  value;          //  Floating-point to support required range.
                            //  A 64-bit integer, if available, is a more
                            //    efficient alternative.
public:
static  double scale;       //  Smallest fraction of monetary unit
                            //    represented e.g. 100 = cents, 1000 = mils
                            //  (Default is 100 -- user can override.)

//   External representation: Constants used in output and input functions
//   ----------------------- (initialized in Money.cpp -- user can override)
public:
 static char   pfx_symbol[];   // Leading currency symbol (U.S.: "$")
 static char   sfx_symbol[];   // Trailing currency symbol (U.S.: "")
 static char   decimal_point;  // Character for 100ths  (U.S.:  period)
 static char   group_separator;// Character for 1000nds (U.S.:  comma)
 static char   unit_name[];    // Name of monetary unit (U.S.:  "dollar")
 static char   cent_name[];    // Name of fraction unit (U.S.:  "cent")
```

```
// Constructors:  To support literal constants, we allow conversion from
// ------------   float.  This inhibits detection of some mixed expressions.


private: static double round(DOUBLE x); // Assures exact conversion from float
public:
   Money(DOUBLE x) : value(round(x * scale)) {}
   Money() {}                            // Default constructor for efficiency

//  The compiler will supply appropriate versions of:
//    -  the destructor,
//    -  the copy constructor,
//    -  the assignment operator.


//  Accessor functions to separate whole and fractional parts:
//  ----------------------------------------------------------
public:
  short   cents()   const
     {double dummy; return short(modf((value + (value < 0 ? -.5 :.5))
                                          / scale, &dummy) * 100);}
  double  wholeUnits()     const
     {double dummy; return modf(double(value) / scale, &dummy),dummy;}


//  Additive pattern arithmetic operators
//  -------------------------------------

#define Class Money
  #include "Additive.hpp"
#undef Class
```

```
//  Relational member operators: (others in global.hpp)
//  ---------------------------

 bool operator== (MONEY rs)   const  {return value == rs.value;}
 bool operator<  (MONEY rs)   const  {return value <  rs.value;}

 bool operator== (DOUBLE rs)  const  {return value == rs * scale;}
 bool operator<  (DOUBLE rs)  const  {return value <  rs * scale;}
friend
 bool operator<  (DOUBLE ls, MONEY rs){return ls*scale <  rs.value;}

};                  //   **********  End of class definition

inline bool  operator== (DOUBLE ls, MONEY rs){return rs == ls;}

ostream& operator<< (ostream& ls, MONEY rs);

#endif
```

```
//  IDI Money class:  implementation of non-in-line functions
//                    and class constants
//    Copyright 1995, Information Disciplines, Inc., Chicago

//  See Money.hpp for documentation and declarations


#include <iostream.h>
#include "Money.hpp"


//  The static member data are public, allowing user overrides without
//  incurring the overhead of member functions.  This is acceptable,
//  because any user errors will be immediately obvious, and are
//  unlikely to affect computational integrity.


// Initial (default) constant values for U.S. currency
// ---------------------------------------------------

//  The user-program may override these values.  To avoid inconsistencies,
//  this should be done only before any Money objects are created.

char       Money::pfx_symbol[]    = "$";
char       Money::sfx_symbol[]    = "" ;
char       Money::decimal_point   = '.';
char       Money::group_separator = ',';
char       Money::unit_name[]     = "dollar";
char       Money::cent_name[]     = "cent";
double     Money::scale           = 100;        //  Cents




double Money::round(const double x)           //  Static internal function
 {double dummy;                                //    to round and truncate
  return modf(x + (x<0 ? -.5 : .5),&dummy), dummy;
 }
```

```
//  Primitive arithmetic member operators:
//  --------------------------------------

//  For efficiency we follow Scott Myers ("More Effective C++", Addison
//  Wesley) in defining the compound assignment operators as primitive.

 Money& Money::operator+= (MONEY        rs) {value += rs.value;      return *this;}
 Money& Money::operator-= (MONEY        rs) {value -= rs.value;      return *this;}
 Money& Money::operator*= (DOUBLE       rs) {value =round(value*rs); return *this;}
 Money& Money::operator/= (DOUBLE       rs) {value =round(value/rs); return *this;}
 double Money::operator/  (MONEY        rs) const {return value/rs.value;}
 Money  Money::operator- ()   const
                  {Money  result;  result.value = - value;  return result;}




//  Output display (stream insertion) operator
//  ------------------------------------------

//  This version displays a Money object in the form:

//   -  leading minus sign, if negative
//   -  floating prefix currency symbol (if symbol_pfx = 1)
//   -  whole amount in groups of three digits separated by punctuation
//   -  decimal point
//   -  2-digit (or more when needed) decimal fraction

  ostream& operator<< (ostream& ls, MONEY rs)
  {Money absx = abs(rs);                  //  Get magnitude of argument
   double whole = absx.wholeUnits();      //  Isolate whole monetary units
   short  cents = absx.cents();           //  Isolate fractional units
   Money  remdr = absx - Money((whole * 100 + cents) / 100);

   if  (rs < 0) ls << '-';                //  Print prefix minus, if needed
   ls << Money::pfx_symbol;               //  Insert dollar sign

//  Continued on next page
```

```
//  Output stream insertion operator function (continued)


//  Print groups of 3 digits separated by punctuation
//  -------------------------------------------------

   const float group_divisors[6] = {1E0f, 1E3f, 1E6f, 1E9f, 1E12f, 1E15f};
   short  grpNum = (whole == 0) ? 0 : short(log10(whole) / 3);
   int    grpVal =  int(whole / group_divisors[grpNum]);

   ls << grpVal;               // Print leftmost 3-digits (no leading 0's)
   while (grpNum != 0)         // For remaining 3-digit groups
   {ls << rs.group_separator;                    // Print group separator
    whole -= grpVal * group_divisors[grpNum--];  // Compute new remainder
    grpVal  = int(whole / group_divisors[grpNum]);// Get next 3-digit value

    if   (grpVal < 100) ls  << '0';              // Insert embedded 0's
    if   (grpVal < 10)  ls  << '0';              //   as needed
    ls << grpVal;}                               // Print 3-digit value

//  Print cents portion
//  -------------------

  ls <<  rs.decimal_point             //  Append decimal point
     << (cents < 10 ? "0" : "")       //  Append leading 0 if needed
     <<  cents;                       //  Append cents value

//  Append any additional fractional digits
//  ---------------------------------------

  for (int i = int(Money::scale/100 -1); i&&(remdr>0); i--, remdr/=10)
       ls << (10. * remdr).cents();
  ls << Money::sfx_symbol;            //    Insert trailing currency symbol

  return ls;}                         //    Allow nested stream operations
```

```
// Temperature and Temperature Change classes (Copyright 1994,
// -------------------------------       Information Disciplines, Inc.)

//  These classes support operations on temperatures.

#ifndef TEMPERATURE
#define TEMP_CHANGE const TempChange
#define TEMPERATURE const Temperature

//   Temperature Change (extent) class
//   ------------------

class TempChange {

friend class Temperature; // Allow Temperature methods to retrieve value

    double value;         // Internal representation degrees Kelvin or Celsius



//  Constructor
//  -----------

public:
    TempChange(DOUBLE x = 0) : value(x) {}

//    Compiler will supply acceptable copy constructor, destructor,
//    and assignment operator




//  Arithmetic Operators
//  --------------------

#define Class TempChange
#define PureType double
#include "Additive.hpp"     /*  Additive pattern for binary operators */
```

```
//  Relational operators
//  --------------------

 bool operator== (TEMP_CHANGE rs) const {return value == rs.value;}
 bool operator<  (TEMP_CHANGE rs) const {return value <  rs.value;}



//  I-O operations
//  --------------

 static char*  unit;        //  Unit name ("degree") for messages
 ostream& put(ostream& s) const;
 istream& get(istream& s);

};               //  ************   End of class definition



//  Inline member and non-member functions
//  --------------------------------------

inline ostream& operator<< (ostream& ls, TEMP_CHANGE rs) {return rs.put(ls);}
inline istream& operator>> (istream& ls, TempChange& rs){return rs.get(ls);}

inline TempChange& TempChange::operator+=(TEMP_CHANGE rs)
                      {value += rs.value; return *this; }
inline TempChange& TempChange::operator-=(TEMP_CHANGE rs)
                      {value -= rs.value; return *this; }
inline TempChange& TempChange::operator*=(DOUBLE rs)
                      {value *= rs;       return *this; }
inline TempChange& TempChange::operator/=(DOUBLE rs)
                      {value /= rs;       return *this; }
inline double TempChange::operator/ (TEMP_CHANGE rs) const
                      {  return value / rs.value; }
```

```
//   Temperature class
//   ----------

//  This class defines Temperature objects.


class Temperature  {
    double  value;              //  Internal representation
public:
    static DOUBLE zeroCelsius;



//  Constructor, accessors, and conversion functions
//  -----------------------------------------------

   Temperature(double kelvin=0.0) : value(kelvin) {assert(value>=0.0);}

//   Compiler will supply acceptable copy constructor, destructor,
//   and assignment operator

       double      toKelvin()               {return value;}
       double      toCelsius()              {return value - zeroCelsius;}
       double      toFahrenheit()           {return 1.8*toCelsius()+32;}
    inline static
       Temperature fromCelsius   (DOUBLE x) {return x+zeroCelsius;}
    inline static
       Temperature fromFahrenheit(DOUBLE x) {return fromCelsius((x-32)/1.8);}

//  Arithmetic Operators

#define PointClass Temperature
  #include "PointExt.hpp"
```

```
//  Relational operators
//  --------------------

 bool operator== (Temperature rs) const {return value == rs.value;}
 bool operator<  (Temperature rs) const {return value <  rs.value;}



//  I-O operations
//  --------------

static char*  unit;        //  Unit name ("K") for messages
ostream& put(ostream& s) const;
istream& get(istream& s);

};


//  Inline member and non-member functions
//  --------------------------------------
inline ostream& operator<< (ostream& ls, Temperature rs) {return rs.put(ls);}
inline istream& operator>> (istream& ls, Temperature& rs){return rs.get(ls);}

inline Temperature& Temperature::operator+= (TEMP_CHANGE rs)
                           {assert(value >= -rs.value); value += rs.value; return *this;}
inline Temperature& Temperature::operator-= (TEMP_CHANGE rs)
                           {assert(value >=  rs.value); value -= rs.value; return *this;}


#undef Class
#undef PointClass
#endif
```

```
// Temperature Implementation
//                          See IDITEMPERATURE.HPP for documentation

#include "global.hpp"
#include "Temperature.hpp"

//  TempChange implementation
//  ----------

 char* TempChange::unit = "degree";

 ostream& TempChange::put(ostream& ls) const
 {ls << value << ' ' << TempChange::unit;
  if (value != 1.0) ls << 's';              //  Append English Plural
  return ls;
 }

 istream& TempChange::get(istream& ls)
 {ls >> value; return ls; }          //  (mainly for debugging)



//  Temperature implementation
//  -----------

 char* Temperature::unit = "\370K";
  DOUBLE Temperature::zeroCelsius = 273.15;

 ostream& Temperature::put(ostream& rs) const
 {rs << value << unit; return rs; }

 istream& Temperature::get(istream& rs)
 {  return rs;
 }
```