Procedural vs. Object-Oriented Programming

- Procedural Programming
  - top down design
  - create functions to do small tasks
  - communicate by parameters and return values
- Object-Oriented Programming
  - design and represent objects
  - determine relationships between objects
  - determine attributes each object has
  - determine behaviours each object will respond to
  - create objects and send messages to them to use or manipulate their attributes

Object-Oriented Programming

- An **object** is a model of a real or imaginary object (e.g. bank account, address),
- Each object is defined by **attributes** (data members) and **operations** (member functions/method).
- A **class** is a set of objects with the same properties.
- A **client** may create a class object and operate on it through a set of operations that are declared public.
- Each object maintains the **state** of its private data structure, and responds to clients by executing code that implements a particular method.
- During the execution of a program, objects are created and operated on, and the application program solves the problem by sending messages to various objects.

Examples of Classes

- name, address, person, employee, student
- die, spinner, card, race car, robot
- clock, timer, elevator
- bank account, time card, transcript
- date, time
- point, vector, matrix, fraction, complex, term, polynomial
- inventory (grocery, bookstore, auto parts)
- book, CD, DVD, journal
- house, room, hotel

Classes

- A class is a *blueprint* for an object—it defines its *attributes* including *data members* and *methods*
- We have used some classes: `string` and `iostream`
- We can define our own classes—we need to include
  1. Data
  2. Methods—functions which access/modify data members

Examples

- Example: `name` class which has two attributes `firstName` and `lastName`
- For `date` we can require `month`, `day` and `year`
- These could be encapsulated into a `date` class
  1. Data would be 3 integers `day`, `month`, `year`
  2. Methods could be `read date`, `print date`, `print date using month name`, `assign one date to another`, `increment date by days`, `increment date by months`, `increment date by years`, `compare same date`, `compare date comes before`, `compare date comes after`, `get month`, `get day`, `get year`, `set month`, `set day`, `set year`, `get todays date`, `check if a year if leap year`, etc.

Methods

- Methods fall into two categories
  1. *Accessors*: retrieve the values of the data members
  2. *Modifiers*: modify the values of the data members
- Consider the `fraction`—we had to use two integers to represent it—we could encapsulate it into a class with data members `numerator` and `denominator` and methods `read`, `print`, `assign`, `get numerator`, `get denominator`, `add`, `sub`, `mult`, `div`, `reduce`, etc.

Constructors

- In addition to the methods we have talked about we also need a way to build (and initialize) an object
- The methods to do this are called *constructors*
- Two kinds of constructors
  1. *Default constructor*—constructor with no arguments— data members are initialized with default values
  2. *Constructor with arguments*—data members are initialized with the input arguments

Constructors

Constructor rules

1. If no constructor—data members are not initialized
2. If no parameter—call default constructor
3. It is generally a good practice to always provide a default constructor
4. If any non-default constructor is defined, default constructor is not implicitly defined.

Operators

Note that the C++ standard operators do not work with user-defined objects unless we define them to do so (operator overloading). So we cannot use + or == with fractions or ++ or == with dates.

However there are two exceptions

1. The assignment operator works: it just assigns each data member of an object
2. The dot operator works: it allows us to access the class members

Class Construction

- Classes consist of two parts
  1. The interface: contains the declaration of the class including the data members and methods
  2. The implementation: contains the definition of the methods
- It could be one file but typically is two files
  1. header—interface—`.h` file
  2. implementation—definition—`.cc` or `.cpp` file
- We will need to include (`#include "header-file.h"`) whenever we want to use objects of the class it defines

Client Program

- The program we write that uses the class is called the *client program*
- So far we have been writing clients only. All our programs used `iostream` class objects. Many have used `string` class objects
- We will switch gears a little. We will now spend most of our time writing classes and the client will just be a driver to test the classes and their methods

Class Definition

- The *public section* (`public:`) of a class consists of the specification of any variables, types, constants, and function prototypes that are accessible to another program component
- The *private section* (`private:`) of a class consists of the specification of any variables, types, constants, and function prototypes to be hidden from other program components
- A record (`struct`) is a class in which all members and methods are declared `public`.

Information Hiding

- We should declare data members private unless we have a good reason to do otherwise.
- Access to the data members should be done through accessor and modifier methods only.
- Debugging is easier if data members can only be changed by the members defined by the class itself.
- Users of the class do not need to know the internal representation of the data (easier to use).
- Implementation of the class can change without affecting its users (as long as the interface is the same).

Compiler Directives

- Sometimes, a number of classes all use other classes.
- Chains of `#include` may end up including a header file multiple times.
- A class will be defined multiple times.

```
#ifndef CLASS_NAME_H
#define CLASS_NAME_H

// class definition

#endif
```

Class Implementation

- The **implementation file** contains the C++ code for the class member functions
- With the **scope resolution operator** `::` as a prefix to the function name in each function header
- The implementation file contains the following member functions
  1. Constructors that execute when an object of a class is declared and that set the initial state of the new object
  2. Accessors that retrieve the value of a data member
  3. Modifiers that modify the value of a data member
- Notice that the implementation file contains

```
#include "class-name.h"
```

Use of Classes and Objects

- The client program (i.e., program component) contains the declaration

  `class-name object;`

- Any program component may apply member functions that are `public` as operations on the declared object
- Member functions that are `private` are not directly accessible outside the class
- The *client* program may declare and manipulate objects of the data type defined by the *server* class
- Notice that the client program contains

  `#include "class-name.h"`

Classes as Operands and Arguments

- If `class-name` is a class type, use `class-name& object` to declare `object` as a formal reference parameter
- If `class-name` is a class type, use `const class-name& object` to specify that formal parameter `object` cannot be changed by the function's execution
- Accessor methods should be `const`

Static Members and Methods

- Normally, data members and methods are associated with each object. Different instances of a class have separate copies.
- Static data members are associated with the class, not to individual instances.
- Similarly, static methods are associated with the class.
- To declare static members and methods, add `static` in front of the declaration.
- To refer to a static member or to call a static method, use `class-name::member` or `class-name::method()`.
- Useful for constants related to the class, or methods not associated with a particular object.

Name Class: Interface (name.h)

```
// Interface file for a simple name class
#ifndef NAME_H
#define NAME_H
using namespace std;

class Name {
 public:
   // constructors
   Name();                              // default constructor
   Name(const string& surname);         // given name will be enpty
   Name(const string& given, const string& surname);

   // input/output methods
   void read();                         // read in a name in form given last
   void print() const;                  // print in format given last
   void revPrint() const;               // print in format last, given
```

```
   // set methods
   void setSurname(const string&);  // change the surname
   void setGiven(const string&);    // change the given name
   void setName(const string& given, const string& surname); // whole name

   // get methods
   string surname() const;          // return the surname
   string givenName() const;        // return the given name
   string initials() const;         // return the initials

   // comparison methods
   bool sameAs(const Name&) const;  // is the name the same as this one
   bool isBefore(const Name&) const;// does this name come before the
                                    // param telephone directory style
                                    // i.e. lastname, firstname
 private:
   // data members
   string last;
   string first;
};                                  // ';' is required
#endif
```

Name Class: Implementation (name.cc)

```
// implementation of the name class

// preprocessor directives
#include <iostream>
#include <string>
#include "name.h"          // class definition for this class

using namespace std;

//*************************************************************
// default constructor
// post-condition -- last name will be "UNKNOWN"  and the given name
//                   will be the empty string
//*************************************************************
Name::Name() {
   last = "UNKNOWN";
   first = "";
}
```

```
//*********************************************************
// constructor to set the last name
// parameter usage : surname imports the name to use as the last name
// post-condition -- last name will be the surname and the given name
//                   will be the empty string
//*********************************************************
Name::Name(const string& surname) {
   last = surname;
   first = "";
}
//*********************************************************
// constructor to set both names
// parameter usage : surname -- imports the name to use as the last name
//                   given -- imports the name to use as the first name
// post-condition -- last name will be the surname and the given name
//                   will be given
//*********************************************************
Name::Name(const string& given, const string& surname) {
   last = surname;
   first = given ;
}
```

```
//*************************************************************
// method to read a name
// post-condition -- the name will be set to the input value
//*************************************************************
void Name::read() { cin >> first >> last; }

//*************************************************************
// method to print a name
//*************************************************************
void Name::print() const { cout << first << ' ' << last; }

//*************************************************************
// method to print a name in format surname, givenName
//*************************************************************
void Name::revPrint() const { cout << last << ", " << first; }
//*************************************************************
// method to change the surname
// parameter usage : surname -- imports the name to use as the last name
// post-condition -- last name will changed to surname
//*************************************************************
void Name::setSurname(const string& surname) { last = surname; }
```

```
//*************************************************************
// method to change the given name
// parameter usage : given -- imports the name to use as the first name
// post-condition -- first name will changed to given
//*************************************************************
void Name::setGiven(const string& given) { first = given; }
//*************************************************************
// method to change both names
// parameter usage : surname -- imports the name to use as the last name
//                   given -- imports the name to use as the first name
// post-condition -- last name will changed to surname and
//                   first name will changed to given
//*************************************************************
void Name::setName(const string& given, const string& surname) {
   last = surname;  first = given;
}
//*******************************
// method to return the surname
//*******************************
string Name::surname() const { return last; }
```

```
//*********************************
// method to return the given name
//*********************************
string Name::givenName() const { return first; }
//*************************************************************
// method to return the initials
// post-condition -- a string containing the first letters of the name
//                   will be returned
//*************************************************************
string Name::initials() const {
   string temp = "";
   if (first != "") temp += first[0];     // add first initial
   if (last != "UNKNOWN") temp += last[0];// add last initial
   return temp;
}
//*************************************************************
// method to determine is the parameter is the same as this name
//
// parameter usage : name imports the name to compare to this one
// post-condition -- true is returned if the parameter is exactly
//                   the same as this name
//*************************************************************
```

```
bool Name::sameAs(const Name& name) const {
   return first == name.first && last == name.last;
}

//*************************************************************
// method to determine if this name comes before the parameter
//
// parameter usage : name imports the name to compare to this one
// post-condition -- true is returned if the parameter alphabetically
//                   comes before this name when considered last name
//                   first
//*************************************************************
bool Name::isBefore(const Name& name) const {
   return last < name.last || last == name.last && first < name.first;
}
```

Name Class – Client Program

```
//*************************************************************
// client testing program for the Name class
//*************************************************************

#include <iostream>
#include <string>
#include "name.h"

using namespace std;
```

Name Class: Client Program

```
int main() {
   // test the three constructors
   cout << "Testing the 3 constructors" << endl;
   Name name1;
   Name name2("Bennett");
   Name name3("Sophie Louise", "MacGregor");
   Name name4 = name3;

   cout << "The constructed names are : " << endl;
   name1.revPrint(); cout << endl;
   name2.revPrint(); cout << endl;
   name3.revPrint(); cout << endl;
   cout << "Assignment operator test." << endl;
   cout << "After name4 = name3;";
   cout << " name 4 is "; name4.revPrint(); cout << endl << endl;

   return 0;
}
```

Date Class: Interface (date.h)

```
//interface file for a simple date class
#ifndef DATE_H
#define DATE_H

using namespace std;

class Date
{
 public:
   // constructors
   // default values are yr -- 2000; mon -- 1; day -- 1;
   // default values will be used if no parameter or parameter is invalid
   Date();                          // default constructor set to 2000/1/1
   Date(int yr);                    // set to yr/1/1
   Date(int yr, int mon);           // set to yr/mon/1
   Date(int yr, int mon, int dd);   // set to yr/mom/dd

   // input/output methods
```

```
void read();           // read in a date
                       // acceptable formats are
                       //    yyyy/mm/dd
                       //    yyyy-mm-dd
                       //    yyyy:mm:dd
                       // if format is incorrect or the
                       // date is invalid, date is not changed
void print() const;    // print in format yyyy/mm/dd
void printWithWord() const;// print in format month dd, yyyy

// set methods -- date is left unchanged if parameters are not valid
bool setYear(int);                  // set the year
bool setMonth(int);                 // set the month
bool setDay(int);                   // set the day

// get methods -- return the values of the private data members
int year() const;
int month() const;
int day () const;
// increment methods
void addYear(int);                  // add int years to the date
void addMonth(int);                 // add int months to the date
```

```
   void addDay(int);                    // add int days to the date

   // comparison methods
   bool sameAs(const Date&) const;
   bool isBefore(const Date&) const;

   // static method -- for the class
   // not available to a specific instance
   static bool leapYear(int yr);  // is the parameter a leap year?

 private:
   int yy; int mm; int dd;

   // private helper functions
   bool setDate(int yr, int mon, int dd);// set the date to this date
   static bool validDate(int yr, int mon, int day);
   static int daysIn(int mon, int yr); // returns # of days in the month
};                                     // ';' is required
#endif
```

Date Class: Implementation (date.cc)

```
// preprocessor directives
#include <iostream>
#include <string>
#include "date.h"          // class definition for this class

using namespace std;

// lowest valid year for this type of calendar
const int baseYear = 1582;

// an array of strings which are the names for each month;
// used to simplify the printing of the month name
string monthName[] = {"Jan","Feb","Mar","Apr","May","June",
                      "July","Aug","Sept","Oct","Nov","Dec"};

//*************************************************************
// constructor to initialize to 2000/1/1
// post-condition -- the date will be initialized to the default
//*************************************************************
Date::Date() { yy = 2000;  mm = 1; dd = 1; }
```

```
//*****************************************************************
// constructor to initialize to yr/1/1
// parameter usage : yr -- imports the year to use for the date
// post-condition -- if the year is valid the date will be yr/1/1
//                   otherwise it will be 2000/1/1
//*****************************************************************
Date::Date(int yr) {
   if (yr >= baseYear) yy = yr; else yy = 2000;
   mm = 1; dd = 1;
}
//*****************************************************************
// constructor to initialize to yr/mon/1
// parameter usage : yr -- imports the year to use for the date
//                   mon -- imports the month to use for the date
// post-condition -- if the year and month are valid the date will be yr/mon/1
//                   otherwise it will be 2000/1/1
//*****************************************************************
Date::Date(int yr, int mon) {
   dd = 1;
   if (validDate(yr, mon, dd)) {
       yy = yr; mm = mon;    // set date to the entered date
    } else {                 // date was not valid so use default values
       yy = 2000; mm = 1; }
}
```

```
//*************************************************************
// constructor to initialize to yr/mon/day
//
//  parameter usage : yr -- imports the year to use for the date
//                    mon -- imports the month to use for the date
//                    day -- imports the day to use for the date
//
// post-condition -- if the parameters are valid the date will be yr/mon/day
//                   otherwise it will be 2000/1/1
//*************************************************************
Date::Date(int yr, int mon, int day) {

    if (validDate(yr, mon, day)) {
       yy = yr;              // set date to the entered date
       mm = mon;
       dd = day;
    }
    else {                   // date was not valid so use default values
       yy = 2000;
       mm = 1;
       dd = 1;
    }
}
```

```
//*************************************************************
// set the  year of this object to yr
//
//  parameter usage : yr -- imports the year to change to
//
// post-condition -- if yr is valid the date's year will be set to yr
//                   and true will be returned.  If invalid, no change is
//                   made and false is returned
//*************************************************************
bool Date::setYear(int yr) { return setDate(yr,mm,dd); }

//*************************************************************
// set the month of this object to mon
//
//  parameter usage : mon -- imports the month to change to
//
// post-condition -- if moh is valid the date's month will be set to mon
//                   and true will be returned.  If invalid, no change is
//                   made and false is returned
//*************************************************************
bool Date::setMonth(int mon) { return setDate(yy,mon,dd); }
```

```
//*************************************************************
// set the day of this object to day
//
//  parameter usage : day -- imports the day to change to
//
// post-condition -- if  day is valid the date's day will be set to day
//                   and true will be returned.  If invalid, no change is
//                   made and false is returned
//*************************************************************
bool Date::setDay(int day) { return setDate(yy,mm,day); }

//*************************************************************
// function to return this objects year
//
// post-condition -- the year is returned
//*************************************************************
int Date::year() const { return yy; }

//*************************************************************
// function to return this objects month
//
// post-condition -- the month is returned
//*************************************************************
int Date::month() const { return mm; }
```

```
//*************************************************************
// function to return this objects day
//
// post-condition -- the dayis returned
//*************************************************************
int Date::day() const { return dd; }

//*************************************************************
// function to add a specified number of years to his ojects date.
//
// parameter usage :
//  numYrs -- imports the number of years to add to this objects date.
//
// post-condition -- the date is adjusted by numYr; if numYrs > 0
//                   they are added to the date, if < 0 they are subtracted
//                   from the date.
//*************************************************************
void Date::addYear(int numYrs) {

   yy += numYrs;

   // in a non leap year we can not have Feb 29 so change it to the 28th
   if (!leapYear(yy) &&  mm == 2 && dd == 29) dd = 28;
}
```

```
//*************************************************************
// function to add a specified number of months to his ojects date.
// parameter usage : numMonths -- imports the number of months to add
// post-condition -- the date is adjusted by numMonths; if numMonths > 0 they are
//                   added to the date, if < 0 they are subtracted from the date.
//*************************************************************
void Date::addMonth(int numMonths) {

   mm += numMonths;

   while (mm > 12) {   // while more months than in a year
      mm -= 12;        //    reduce number of months and
      yy++;            //    adjust to next year
   }

   while (mm < 1)  {   // while less months than in a year
      mm += 12;        //    increase number of months and
      yy--;            //    adjust to prev year
   }

   // adjust end of month if dd is past end of current month this will ensure
   // that the date is always valid, i.e we do not end up with Feb 30 or something
   if (dd > daysIn(mm,yy)) dd = daysIn(mm,yy);
}
```

```
//*********************************************************************
// function to add a specified number of days to this objects date.
// parameter usage : numDay -- imports the number of days to add
// post-condition -- the date is adjusted by numDays; if numDays > 0 they are
//                   added to the date, if < 0 they are subtracted from the date.
//*********************************************************************
void Date::addDay(int numDays) {
   dd += numDays;
   while (dd > daysIn(mm,yy)) {// while more days than in current month
      dd -= daysIn(mm,yy);     //   subtract number of days in current month
      mm++;                    //   go to next month
      if (mm > 12) {           //   check for new year and adjust if necessary
         mm = 1; yy++;
      }
   }
   while (dd < 1) {            // while fewer days than in a month
      mm--;                    //   go to prev month
      if (mm < 1) {            //   check for new year and adjust if necessary
         mm = 12;
         yy--;
      }
      dd += daysIn(mm,yy);      // add number of days in this month
   }
}
```

```
//*************************************************************
// function to return true if the date is the same as the date argument
//
// parameter usage : date -- imports the date to be compared with
//
// post-condition -- returns true if the same date
//*************************************************************
bool Date::sameAs(const Date& date) const {

   return (dd == date.dd && mm == date.mm && yy == date.yy);
}
//*************************************************************
// function to return true if the date comes before the date argument.
//
// parameter usage : date -- imports the date to be compared with
//
// post-condition -- returns true if it comes before
//*************************************************************
bool Date::isBefore(const Date& date) const {

   return (yy < date.yy ||
           (yy == date.yy && mm < date.mm) ||
           (yy == date.yy && mm == date.mm && dd < date.dd));
}
```

```
//*****************************************************************
// function to return true if year is a leap year.
//
//  parameter usage : year -- imports the year to evaluate
//
// post-condition -- returns true if it is a leap year
//*****************************************************************
bool Date::leapYear(int year) {
   return (year % 400==0 || year % 4 == 0 && year % 100 != 0);
}

//*****************************************************************
// print the date in form yyyy/mm/dd
//*****************************************************************
void Date::print() const {
   cout << yy << '/' << mm << '/' << dd;
}

//*****************************************************************
// print the date in form mon dd, yyyy
//*****************************************************************
void Date::printWithWord() const {
   cout << monthName[mm-1] << ' ' << dd << ", " << yy;
}
```

```
//*************************************************************************
// function to read a date.
//*************************************************************************
void Date::read() {
   int day,mon,yr;   char sep1,sep2;

   cin >> yr >> sep1 >> mon >> sep2 >> day;
   if (cin.good() && (sep1 == '/' || sep1 == ':' || sep1 == '-') && sep2 == sep1)
      if (!setDate(yr,mon,day)) {
         cout << "Invalid date.  Date has not been changed." << endl;
      }
   else {  // bad data found -- warn
      cout << "Invalid date format.  Date has not been changed." << endl;
      cin.clear();  // clear the stream error flag
   }
}
```

```
//*************************************************************
// utility helper functions
// set the date to the parameter values
//
//  parameter usage : yr -- imports the year to use for the date
//                    mon -- imports the month to use for the date
//                    day -- imports the day to use for the date
//
// post-condition -- if the date is valid the date will be set to it
//                   and true will be returned.  If invalid, no change is
//                   made and false is returned
//*************************************************************
bool Date::setDate(int yr, int mon, int day) {

   if (validDate(yr,mon,day)) {        // if valid change it otherwise do not
      yy = yr;
      mm = mon;
      dd = day;
      return true;
   }
   return false;
}
```

```
//*************************************************************************
// function to determine if a date is valid
//
//  parameter usage : yr -- imports the year to check
//                    mon -- imports the month to check
//                    day -- imports the day to check
//
// post-condition -- returns true if the date is valid
//*************************************************************************
bool Date::validDate(int year, int mon, int day) {
    return (year >= baseYear && (mon >= 1 && mon <= 12) &&
            (day > 0 && day <= daysIn(mon,year)));
}
```

```
//*************************************************************
// function to determine the number of days in a given month
//
//  parameter usage : yr -- imports the year to use
//                    mon -- imports the month to use
//
// post-condition -- returns the number of days in the mon for the given yr
//*************************************************************

int Date::daysIn(int mon, int yr) {
   switch(mon) {
      case 4  :
      case 6  :
      case 9  :
      case 11 : return 30;
      case 2  : if (mon == 2 && leapYear(yr)) return 29; else return 28;
      default : return 31;
   }
}
```

Date Class: Client Program

```
//*************************************************************
// Driver program to test the date class
//*************************************************************
#include <iostream>
#include "date.h"

using namespace std;

int main()
{
    // get valid user input
    Date day0;
    cout << "Enter an valid date in the format yyyy/mm/dd -- ";
    day0.read();
    cout << "After valid input day 0 is ";
    day0.print();
    cout << endl;

    // get invalid user input
    cout << "Enter an invalid date in the format yyyy/mm/dd -- ";
    day0.read();
    cout << "After invalid input day 0 is ";
```

```
day0.print();
cout << endl;

// initiallize date with no parameters -- result should be the default
Date day1;
cout << "after declaration with no parameters " << "day1 is ";
day1.print();
cout << endl;

// check assignment of a date
day1 = day0;
cout << "after assignment to day 0, day 1 is ";
day1.print();
cout << endl;

// create a valid date
Date day2{2000,2,29};
cout << "After Date day(2000,2,29), day 2 is ";
day2.printWithWord();
cout << endl;

// createing an invalid date
Date day3{1997,9,31};       // not 31 days in  9th month
cout << "After Date day3(1997,9,31), day 3 is ";
day3.printWithWord();
cout << " since Sept 31 is invalid. " << endl;
```

```
// set the year valid
day2.setYear(1992);
cout << "After setting year to 1992, day2 is ";
day2.print();
cout << endl;

// set the year invalid
day2.setYear(1997);
cout << "After setting year to 1997, day2 is ";
day2.printWithWord();
cout << " since 29 Feb 1997 is invalid. " << endl;


// set the month valid
day2.setMonth(7);
cout << "After setting month to July, day2 is ";
day2.print();
cout << endl;

// set the day valid
day2.setDay(31);
cout << "After setting day to 31, day2 is ";
day2.print();
cout << endl;
```

```
// set the month invalid
day2.setMonth(6);
cout << "After setting month to Jun, day2 is ";
day2.print();
cout << " since 1992/6/31 is invalid. " << endl;

// set the month valid
day3.setMonth(2);
cout << "After setting month to Feb, day3 is ";
day3.printWithWord(); cout << endl;

// set the day invalid
day3.setDay(31);
cout << "After setting day to 31, day3 is ";
day3.printWithWord();
cout << " since 31 Feb 2000 is invalid. " << endl;


// set the day valid
day3.setDay(29);
cout << "After setting day to 29, day3 is ";
day3.print();
cout << endl;

// subtracting 4 years from Feb 29 -- should still be Feb 29, 1996
day3.addYear(-4);
```

```
cout << "after subtracting 4 years from day3, it is ";
day3.printWithWord(); cout << endl;

// subtracting 365 days from Feb 29 -- should be Mar 1, 1995
day3.addDay(-365);
cout << "after subtracting 365 days from day3, it is ";
day3.print(); cout << endl;

// subtracting 37 months -- should be 1 Feb 1992
day3.addMonth(-37);
cout << "after subtracting 37 months from day3, it is ";
day3.print(); cout << endl;

Date day4{1996,2,29};
cout << "day4 is ";
day4.print(); cout << endl;

// subtracting 366 days -- should be 28 Feb 1995
day4.addDay(-366);
cout << "after subtracting 366 days from day4, it is ";
day4.print(); cout << endl;

// subtracting 28 days to get to the 31st of Jan 1995
day4.addDay(-28);
cout << "after subtracting 28 days from day4, it is ";
day4.print(); cout << endl;
```

```
// checking that subtracting 2 months from 31 gives end of prev month
// subtracting 2 months -- should be 30 Nov 1994
day4.addMonth(-2);
cout << "after subtracting 2 months from day4, it is ";
day4.print(); cout << endl;

// testing the leapYear function
day4.print(); cout << " is";
if (!Date::leapYear(day4.year()))  cout << " not";
cout << " in a leap year." << endl;


// testing lots of them -- should be 30 Sep 1997
day4.addYear(+2);
day4.addDay(+31);
day4.addMonth(+9);
cout << "after adding 2 years, 31 days and 9 months, day 4 is ";
day4.print(); cout << endl;

// testing comparison operators
day2.print();
if (day2.sameAs(day2)) cout << " is the same as ";
else cout << " is not the same as ";
```

```
    day2.print(); cout << endl;

    day3.print();
    if (day3.sameAs(day2)) cout << " is the same as ";
    else cout << " is not the same as ";
    day2.print(); cout << endl;

    day4.print();
    if (day4.isBefore(day3)) cout << " comes before ";
    else cout << " does not come before ";
    day3.print(); cout << endl;


    day3.print();
    if (day3.isBefore(day4)) cout << " comes before ";
    else cout << " does not come before ";
    day4.print(); cout << endl;

    return 0;
}
```

Fraction Class: Interface (fraction.h)

```
// Interface file for FRACTION class -- fraction.h
#ifndef FRACTION_H
#define FRACTION_H

using namespace std;

class Fraction {
 public:
   // constructors
   Fraction();                    // default constructor -- results in 0/1
   Fraction(int num);             // one parameter  -- results in num/1
   Fraction(int num, int den);    // two parameters -- results in num/den

   // input and output methods
   void read();                // input must be one of the following forms :
                               //      1. integer
                               //      2. integer/integer
                               // Note: no spaces allowed in the fraction
   void print() const;
   void printMixed() const;             // print as a mixed number if the
                                        // fraction is improper
   // accessor methods
```

```
   int numerator() const;                // returns the numerator
   int denominator() const;              // returns the denominator

   // arithmetic methods
   Fraction add(const Fraction& frac) const;  // returns the sum
   Fraction sub(const Fraction& frac) const;  // returns the difference
   Fraction mult(const Fraction& frac) const; // returns the product
   Fraction div(const Fraction& frac) const;  // returns the quotient
   Fraction reciprocal() const;               // returns the reciprocal

   // comparison methods
   bool sameAs(const Fraction& frac) const;    // equality
   bool isBefore(const Fraction& frac) const;  // less than

 private:
    // Data members
    int numer;
    int denom;

    // Function member
    void reduce();               // reduces to lowest terms with denom > 0
};
#endif
```

Fraction Class: Implementation (fraction.cc)

```
// Class implementation file: fraction.cc
#include <cassert>
#include <cstdlib>     // this is needed for the abs function
#include "mymath.h"
#include "fraction.h"
using namespace std;
//*********************************************
// default constructor;  creats 0/1
//*********************************************
Fraction::Fraction() {
   numer = 0;
   denom = 1;
}
//*********************************************
// constructor to create num/1.
// num imports the numerator to use
//*********************************************
Fraction::Fraction(int num) {
   numer = num ;
   denom = 1;
}
```

```
//*********************************************
// constructor to create num/den.
// num imports the numerator to use
// den imports the denominator to use
// pre-condition -- den is not 0
//*********************************************
Fraction::Fraction(int num, int den) {
   assert(den != 0);  // denominator of 0 is invalid so crash
   numer = num ;
   denom = den;
   reduce();           // ensure denominator is positive and fraction reduced
}
//*************************************************************
// method to add this fraction and the argument and return the result
// frac -- imports the fraction to add to this one
//*************************************************************
Fraction Fraction::add(const Fraction& frac) const {
   Fraction sum;
   sum.denom = lcm(denom, frac.denom);
   sum.numer = numer * (sum.denom / denom) +
               frac.numer * (sum.denom /frac.denom);
   sum.reduce();
   return sum;
}
```

```
//*****************************************************************
// method to subtract the argument from this fraction and return the result
// frac -- imports the fraction to subtract from this one
//*****************************************************************
Fraction Fraction::sub(const Fraction& frac) const {
   Fraction diff;
   diff.denom = lcm(denom, frac.denom);
   diff.numer = numer * (diff.denom / denom) -
                frac.numer * (diff.denom /frac.denom);
   diff.reduce();
   return diff;
}
//*****************************************************************
// method to multiply this fraction and the argument and return the result
//
// frac -- imports the fraction to multiply by
//*****************************************************************
Fraction Fraction::mult(const Fraction& frac) const {
   int num = numer * frac.numer;
   int den = denom * frac.denom;

   Fraction prod(num, den);
   return prod;
}
```

```
//*****************************************************************
// method to divide this fraction by the argument and return the result
// frac -- imports the fraction to divide by
// pre-condition -- frac is not 0
//*****************************************************************
Fraction Fraction::div(const Fraction& frac) const {
   return mult(frac.reciprocal());
}

//*****************************************************************
// method to return the reciprocal of this fraction
// pre-condition -- this fraction is not 0
//*****************************************************************
Fraction Fraction::reciprocal() const { return Fraction(denom, numer); }

//*****************************************************************
// method to return the numerator
//*****************************************************************
int Fraction::numerator() const { return numer; }

//*****************************************************************
// method to return the denominator
//*****************************************************************
int Fraction::denominator() const { return denom; }
```

```
//*************************************************************************
// method to compare the argument to this fraction and return true if they
// are the same
//
// frac -- imports the fraction to compare to this one
//*************************************************************************
bool Fraction::sameAs(const Fraction& frac) const {
   return numer == frac.numer && denom == frac.denom;
}

//*************************************************************************
// method to compare the argument to this fraction and return true if this
// fraction is less than the argument
//
// frac -- imports the fraction to compare to this one
//*************************************************************************
bool Fraction::isBefore(const Fraction& frac) const {
   return numer * frac.denom < frac.numer * denom;
}
```

```
//*************************************************************
// private helper function to reduce fraction
// post-condition -- the fraction is reduced to lowest terms and the
//*************************************************************
void Fraction::reduce() {
   if (denom < 0) {       // ensure that denominator is always positive
      numer = -numer;
      denom = -denom;
   }
   // gcd must have positive ints so send absolute value of numerator
   int comDivisor = gcd(abs(numer), denom);
   numer /= comDivisor;
   denom /= comDivisor;
}

//*************************************************
// method to print a fraction
//*************************************************
void Fraction::print() const {
   cout << numer;
   if (numer != 0 && denom != 1) // only print / and denominator
      cout << "/" << denom; // if fraction is not 0 and the denominator is not 1
}
```

```
//*********************************************************
// method to print a fraction as a mixed number if necessary
//*********************************************************
void Fraction::printMixed() const {
   if (numer < denom || denom == 1)    // proper fraction so just print it
      print();
   else                                // improper fraction print as mixed
      cout << numer/denom << '&' << numer%denom << '/' << denom;
}

//*****************************************************************
// method to read a fraction.
//
// post-condition -- if the data entered was valid, the fraction will be
//                   set to it and reduced.  If data entered is not valid
//                   the value of the fraction is undefined and cin will
//                   have the error flag set.
//*****************************************************************
```

```
void Fraction::read() {
   cin >> numer;
   if (!cin.good()) return;    // error so return with error state

   if (cin.peek() != '/')      // no slash coming so no denominator
      denom = 1;                //      so set denominator to 1
   else {                       //
      cin.ignore();            // ignore the slash
      cin >> denom;            // get the denominator
      if (!cin.good()) return; // error so return with error state
      if (denom == 0) {         // denominator is o
         cin.clear(ios::badbit); //    set bad bit -- data lost
         return;                //    return with error state set
      }
   }
   // we have a valid fraction so reduce it
   reduce();
}
```

Fraction Class – Client Program

```
// File: fractionTest.cc
// Tests the fraction class
#include <iostream>
#include "fraction.h"
using namespace std;

int main() {
   Fraction f1, f2;
   Fraction f3;

   // Read two fractions
   cout << "Enter the 1st fraction -- " << endl;
   f1.read();
   cout << "Enter the 2nd fraction -- " << endl;
   f2.read();

   // Display the results of fraction arithmetic
   f3 = f1.mult(f2);
   f1.print(); cout << " * ";
   f2.print(); cout << " = ";
   f3.print(); cout << endl;
```

```
    f3 = f1.div(f2);
    f1.print(); cout << " / ";
    f2.print(); cout << " = ";
    f3.print(); cout << endl;

    f3 = f1.add(f2);
    f1.print(); cout << " + ";
    f2.print(); cout << " = ";
    f3.print(); cout << endl;

    f3 = f1.sub(f2);
    f1.print(); cout << " - ";
    f2.print(); cout << " = ";
    f3.print(); cout << endl;

    return 0;
}
```

Relationships Among Objects

There are three main type of relationships.

Has : an object in one class may "own" objects of another class. Also known as **composition**. e.g. A `computer` has a `keyboard`.

Knows : an object in one class may need to know something about another class. e.g. a `student` knows which `courses` he/she is enrolled in, but the `student` does not own the `courses`.

Is : an object of one class may share some of its characteristics with another class. Also known as **inheritance**. e.g. a `student` is a `person`.

Each relation can be further described by its **multiplicity**. e.g. Each `student` may take many `courses`.

Classes

- Allows us to group items of different types.
- Use dot notation to access members (or `->` for pointers).
- Class objects are usually passed by (constant) reference for efficiency.
- Can hide data or methods (encapsulation).
- Enable code reuse by using previously defined classes.

Constructors

- A constructor function is automatically called every time a variable of the class is declared or created (e.g. by `new`).
- It has the same name as the class.
- The role of constructor is to initialize the object's data members.
- The **default constructor** is a constructor which requires no parameters.
- Constructors can be overloaded.
- If no constructors are defined, the compiler supplies a default constructor which does nothing.
- When an array of objects is declared, the default constructor is called.

Constructors

- It is common to have many different forms of constructors.
- Default parameters can reduce the number of constructors.

  ```
  Fraction(int n = 0, int d = 1);
  ```

  Then

  ```
  Fraction zero[3];                    // 0/1
  Fraction five{5};                    // 5/1
  Fraction *x = new Fraction{-2,3};    // -2/3
  ```

- Note: the default values in the parameters should be given only in the declaration of the function (inside the class definition), not in the implementation.
- The syntax for implementing constructors is almost the same as any other member function.

Implementation of Constructors

- Data members can be initialized by assignment statements as in any other functions, but this is inefficient if the data members are complex classes (more on this later).
- An `initialization list` is preferred.

```
Fraction::Fraction(int n, int d)
  : numer{n}, denom{d} { }
```

- Think of initialization list as calling the constructors for the specified data members.
- The initialization list is executed first before any commands inside `{}` are carried out.
- Constant and reference data members can only be initialized in constructors—you must use initialization list for these.

Example

- Suppose we have a `Time` class which has the constructor

```
Time(int hr = 0, int min = 0, int sec = 0);
```

- If we have a `Lecture` class containing two `Time` data members `start` and `end`. We can write the constructor as:

```
Lecture::Lecture(int start_hr, int start_min,
                 int end_hr, int end_min,
                 const string& room)
  : start{start_hr, start_min}, end{end_hr, end_min},
    room_name{room}
{
   ...
}
```

Copy Constructor

- If we write

```
Time t1{15,0,0};
Time t2 = t1, t3{t1};
```

  then the declaration of `t2` and `t3` will call the **copy constructor** to initialize them.

- The prototype for the copy constructor is

```
<classname>(const <classname>& x);
```

- In this case, we want

```
Time(const Time& x);
```

- The copy constructor is also called when object parameters are passed by value and when objects are returned in a function.
- That is why it is more efficient to pass objects by (constant) reference if possible.

Copy Constructor

- If no copy constructor is defined, a default copy constructor is defined for you by the compiler.
- The default copy constructor simply copies each data member using the assignment operator =.
- In most cases this is what we want.

Copy Constructor

What about for a class like this?

```
Class Array {
private:
  int *p, n;
public:
  Array(int size) : n{size} { p = new int[n]; }
  void change(int m) {
    int *temp;
    n = m;
    temp = new int[m];
    delete[] p;
    p = temp;
  }
};
```

Copy Constructor

- The default copy constructor performs **shallow copying**.
- If an object allocates memory dynamically, you need to do **deep copying**—space has to be allocated and the content has to be copied.

```
Array(const Array& a)
  : n{a.n}
{
  p = new int[a.n];
  for (int i = 0; i < n; i++) {
    p[i] = a.p[i];
  }
}
```

Type Conversion

- We have seen constructors such as

  ```
  Fraction(int n = 0, int d = 1);
  ```

- When we write

  ```
  Fraction x{1};
  ```

  we are **converting** the integer value 1 into an equivalent value of type `Fraction`.

- We can also write `x = Fraction(1);` to perform type conversion.

- In general, type conversion is performed by

  ```
  <typename>(expression)
  ```

  where `<typename>` is the desired type.

Type Conversion

- Even if you write `Fraction x = 1;` the compiler automatically calls the constructor to do type conversion.
- You can use the keyword `explicit` to suppress automatic type conversion.

Destructors

- When an object ceases to exist (e.g. out of scope, delete, etc.), sometimes we need to "clean up". e.g. deleting dynamically allocated memory, closing files, etc.
- The **destructor** is automatically called whenever an object ceases to exist.
- The name of the destructor is `~<classname>()`.
- If a destructor is not defined, the default destructor calls the destructors for each data member.
- If an object contains other objects as data members, the destructors of the data members are automatically called first.
- The destructor cannot be called explicitly.

Destructors

- Back to the `Array` class:

```
Array::~Array()
{
  delete[] p;
  p = NULL;
}
```

- Example:

```
{
  Array a1{10};            // constructor called
  Array *a2;
  a2 = new Array{15};      // constructor called
    ...
  delete a2;               // destructor for *a2 called
}                          // destructor for a1 called
```

Constant Objects

- You may declare an object variable to be constant, just like you can declare an integer constant.

```
const Fraction one{1,1};
```

- If you wish to initialize the constant, you can only do so with the constructors.
- Only accessor functions can be called (those with `const`) at the end of the prototype.
- You can only pass constant object into a function by value or by constant reference.

Static Members

- Static data members are used when something should be shared among all objects of the same class.
- Static member functions are not called for any particular object and can only use static data members.

`this` Pointer

- Every member function in a class `X` is passed an implicit parameter `X *this` which points to the object for which the member function is called.
- Inside a member function, `*this` refer to the object.

```
int Fraction::getNumerator() const
{
  return this->numer;  // same as return numer
}
```

- Why is this useful?

`this` Pointer

- One application: cascading functions. We may wish to do:

```
Fraction x, y, z;
x.add(y).subtract(z).times(y);  /* x = (x + y - z)*y */
```

- If each operation is defined to return a reference to the object, we can cascade. e.g.

```
Fraction& Fraction::add(Fraction y)
{
  numer = numer * y.denom + y.numer * denom;
  denom *= y.denom;
  return *this;        // should remove gcd
}
```

Friends

- In C++, the only functions that may access private members of a class are member functions of the same class, except...
- A class may explicitly grant access to its private members to other functions.

```
class X {
private:
  int a;
public:
  int f();
  friend int g(X x);     // the function g can access x.a
  friend class Y;        // member functions of class Y
                         // can access private members of X
};
```

Friends

- This breaks encapsulation, but is necessary/convenient in some cases.
- Friendship must be explicitly granted and it is one-way. If class `X` grants friendship to class `Y`, class `X` cannot access the private members of class `Y`.

Operators

- Standard operators such as `+`, `=` mean different things for different types. e.g. there is a `-` for integers, one for doubles, and one for pointers.
- We can consider the same operator to be overloaded for different operand types.
- In C++, we can overload standard operators for other types (including user-defined classes).
- It is convenient sometimes to define operators. Makes code more readable when used appropriately.

  Instead of

  ```
  x.add(y).subtract(z).times(y);
  ```

  you can write

  ```
  x = (x + y - z) * y;
  ```

Operators

Operators that can be overloaded:

Arithmetic: `+ - * / % += -= *= /= %= ++ --`

Logical: `! && ||`

Comparison: `== != > < >= <=`

Input/Ouptut: `>> <<` (they have other meanings...)

Assignment: `=`

Misc: `[] ()`

And others...

NOTE: the precedence and associativity rules for the operators do not change even if they are overloaded.

Binary Operators

- A binary operator has two operands. e.g. `x + y`
- Syntax for a binary operator that takes an argument of class X and argument of class Y:

```
<returntype> operator<symbol>(const X& x, const Y& y);
<returntype> X::operator<symbol>(const Y& y) const;
```

- Examples:

```
bool operator==(int y, const Fraction& x);
Fraction Fraction::operator+(const Fraction& y) const;
const Fraction& Fraction::operator+=(int y);
```

Member vs. Non-member Operators

- Member
  - first operand must be an object of the class the operator is defined for
  - first operand is passed implicitly
  - called by `x + y` or `x.operator+(y)`
  - the operators `= [] () ->` must be defined as member functions
- Non-member
  - needs to be `friend` in order to access private members
  - first operand can be of any type
  - called by `x + y` or `operator+(x, y)`
  - the input/output operators `<< >>` must be defined as non-member functions

Temporary Objects

- Some operators such as `+=` change the first operand and return the new value. i.e. return a constant reference to avoid copying.
- Some operators such as `+` produce a new value without changing the original. In this case, allocate a new object and return it.

```
Fraction Fraction::operator+(const Fraction& y) const
{
  Fraction temp{*this};
  temp += y;     // assume += has been defined
  return temp;
}
```

Do not return a reference to temp! (Why not?)

Unary Operators

- Unary operator has only one operand. It is either implicit (member function) or explicit (non-member).
- Examples

```
bool operator!(const Fraction& x);
Fraction Fraction::operator-() const;
```

- called by `x.operator-()` or `-x`

Prefix vs. Postfix Operators

- How do we distinguish prefix and postfix ++?
- Prefix:

```
const Fraction& Fraction::operator++()
{
  numer += denom;
  return *this;        // should remove gcd
}
```

- Postfix:

```
Fraction Fraction::operator++(int)
{
  Fraction temp{*this};
  numer += denom;
  return temp;         // should remove gcd
}
```

Assignment Operator

- The default assignment operator performs shallow copy.
- If deep copy is desired, need to define assignment operator.

```
Array& Array::operator=(const Array& A)
{
  if (this != &A) {     // self assignment is bad!
    delete[] p;
    n = A.n;
    p = new int[n];
    for (int i = 0; i < n; i++) {
      p[i] = A.p[i];
    }
  }
  return *this;
}
```

Assignment Operator

- Why did we check if `this != &A`?
- Should always return a reference so operations can be cascaded.
- Rule of thumb: if you need to write a copy constructor, you probably need to write an assignment operator.

Indexing Operator

- Called by `A[i]`
- Must have one parameter:

```
int& Array::operator[](int i);
const int& Array::operator[](int i) const;
```

- It should return a reference so we can write

```
A[i] = 3;
```

- Should provide a constant version for constant objects.

Input/Output Operators

- We want to write:

```
Fraction x, y;
cin >> x >> y;
cout << x << y << endl;
```

- First operand is a stream, not a `Fraction`. We cannot define operators as member functions.
- Operators have to return a reference to the stream to allow cascading.

Input/Output Operators

```
ostream& operator<<(ostream& os, const Fraction& x)
{
  if (x.denom == 1) {
    os << x.numer;
  } else {
    os << x.numer << '/' << x.denom;
  }
  return os;
}
```

Note that the operator must be a friend of `Fraction` for this to work.

Conversion Operators

- You can also override the cast operator.
- Example:

```
Fraction::operator double() const
{
  return double{numer}/denom;
}
```

- Note that there is no return type defined: it is implicit by the type conversion.
- Called by

```
Fraction x{2,3};
double y = x;     // automatic conversion, y = 0.666...
```

Miscellaneous

- Function call operators: we will look at them later when we discuss function objects.
- Intialization syntax: `()` vs. `{}`.
- Move constructors: implement shallow copying (why?). New C++11 feature.
- C++11 features: member functions marked by `default` and `delete`.