Contents

Finding variance of a set of numeric values
Three ways to create and populate an array of integers
Bubble sort algorithm
Two-dimensional-array example: sales figures
Searching and other array operations
Inheritance with employees and hourly employees
Using bitwise operators and masks to clear and set bits
Examples of classes
Simulating the roll of a die
Testing a file-input operation
Random-access file I/O example
Member functions to display, retrieve, and save one rational-number object
Pseudocode for collections with file I/O

Finding variance of a set of numeric values

Variance is the degree to which a set of numbers differ from the average in a set or collection. A set of numbers with low variance are relatively close together; high variance indicates that the numbers are farther apart. The program below computes variance, which is defined as the average of the absolute difference between each number and the average of the values.

Another measure of how widely data is dispersed is *standard deviation.* This is defined as the square root of the average of the squares of the absolute differences of values from the average value.

```
/* Variance.java:
   Reads a string from file 'update.txt',
   prompts for update of the string,
   updates file.     D. Keil 2/14 */
import java.util.*;
import java.io.*;
import java.lang.Math.*;

public class Variance
{
     public static void main(String[] args)
      throws FileNotFoundException
     {
      // Open input file:
      System.out.println
         ("Reading 'Variance.txt'");
      FileReader reader =
         new FileReader("Variance.txt");
      Scanner fin = new Scanner(reader);

      // Read integers into array:
      int A[] = new int[100];
      int n = 0;
      while (fin.hasNext())
      {
         A[n++]= fin.nextInt();
      }
      fin.close();

      // Display array:
      for (int i = 0; i < n; i++)
      {
         System.out.print(A[i] + " ");
      }
      System.out.println();

      // Find variance:
      int sum = 0;
      for (int i = 0; i < n; i++)
      {
         sum += A[i];
      }
      double avg = (double)sum / n;

      double totVar = 0;
      for (int i = 0; i < n; i++)
      {
         totVar += Math.abs(avg - A[i]);
      }
      double avgVar = totVar / n;
      System.out.printf(
         "totVar = %3.2f\n", totVar);
```

```
    System.out.printf(
       "avgVar = %3.2f\n", avgVar);

    // Find standard deviation:
    double tot_sq_dev = 0;
    for (int i = 0; i < n; i++)
    {
       tot_sq_dev +=
          Math.abs(avg - A[i]) *
          Math.abs(avg - A[i]);
    }
    double avg_sq_dev = tot_sq_dev / n;
    double std_dev = Math.sqrt(avg_sq_dev);
    System.out.printf(
      "tot_sq_dev = %3.2f\n", tot_sq_dev);
    System.out.printf(
       "avg_sq_dev= %3.2f\n", avg_sq_dev);
    System.out.printf(
       "std dev=%3.2f\n ", std_dev);
   }
}
```

Three ways to create and populate an array of integers

The following Java program shows how to declare and initialize an array in Java. The line

```
int A[] = {4,3,7,1,2};
```

creates an array of five integers and initializes its members. The operator *new*, used to allocate memory to objects, is *implicit* in this declaration.

```
// arr3.java
// Initializes and displays an array
// of integers
// D. Keil 2/14

public class Arr3
{
    public static void main(String[] args)
    {
     // Declare and initialize an array
     // of integers:
     int A[] = {4,3,7,1,2};

     // Display it:
     for (int i = 0; i < A.length; i++)
          System.out.print(A[i] + " ");

    }
}
```

An equivalent way to define the array is as follows:

```
final int MAX = 5;
int A[] = new int[MAX];
A[0] = 4;
A[1] = 3;
A[2] = 7;
A[3] = 1;
A[4] = 2;
```

A third way to create and populate an array is from disk:

```
/* arr1.java:
   Reads an array from a file of integers.
   D. Keil 2/14
*/
import java.util.Scanner;
import java.io.FileReader;
import java.io.FileNotFoundException;

public class arr1
{
    public static void main(String[] args)
     throws FileNotFoundException
     {

     // See how big file is:
     int n = countTokensInFile("arr1.txt");

     // Declare and read array from file:
     FileReader reader =
          new FileReader("arr1.txt");
     Scanner fin = new Scanner(reader);
     int A[] = new int[n];
     for (int i = 0; i < n; i++)
     {
          A[i]= fin.nextInt();
     }
     fin.close();

     // Display array:
     for (int i = 0; i < A.length; i++)
     {
          System.out.print(A[i] + " ");
     }
     System.out.println();
    }

    static int countTokensInFile
          (String fname)
     throws FileNotFoundException


    {
     // Open input file:
     System.out.print("Reading "+fname);
     FileReader reader =
          new FileReader(fname);
     Scanner fin = new Scanner(reader);
     int n = 0;
     while (fin.hasNext())
     {
          fin.nextInt();
          ++n;
     }
     fin.close();
     // 'n' stores size of file
     return n;
    }
}
```

Bubble sort algorithm

The Bubble sort algorithm uses a nested loop that scans through an array multiple times, each time swapping each pair of out-of-order values. The result of one pass through the array is that the largest element's value ends up as the last element. To fully sort the array requires up to *n* passes, for arrays of size *n*.

```
/*
 sort.java
 Initializes an array, puts it in ascending
 order using Bubble Sort algorithm.
 CSCI 252 section A, FSU, 3/14
*/
import java.util.*;

class sort
{
   public static void main(String[] Args)
   {
      // initialize and display array:
      int[] A = { 2, 8, 2, 9, 1, 8, 3, 2, 2, 8, 2, 5 };
      for (int i = 0; i < A.length; i++)
      {
         System.out.print(A[i]+" ");
      }
      System.out.println();

      // Sort, using nested loops to repeatedly
      // swap largest element to right (bubble sort):
      for (int j = 0; j < A.length-1; j++)
      {
         for (int i = 0; i < A.length-j-1; i++)
         {
           if (A[i] > A[i+1])
           {
              int temp = A[i];
              A[i] = A[i+1];
              A[i+1] = temp;
           }
      }

      // Display array:
      for (int i = 0; i < A.length; i++)
      {
         System.out.print(A[i]+" ");
      }
      System.out.println();
      }
   }
}
```

Two-dimensional-array example: sales figures

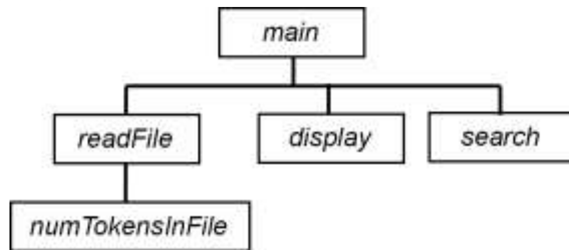
The program below displays a table of sales figures for two products, over four quarters. It shows the use of nested loops in displaying a two-dimensional matrix of integers.

```
/*
  salesarr.java:
  Initializes and displays a two-dimensional
  array of integers.
  D. Keil, 3/14
*/

public class salesarr
{
    public static void main(String[] args)
    {
     // Declare and initialize array:
     int sales[][] = {
          {4,3,7,1},
          {2,2,3,1}
     };

     // Display it using nested loop:
     for (int i = 0; i < 2; i++)
     {
          for (int j = 0; j < 4; j++)
               System.out.print(sales[i][j] + " ");
          System.out.println();
     }

    }
}
```

Searching and other array operations

The *linear search* is a simple loop algorithm that makes one traversal (pass) through an array to find a match with a *search key*. The program below reads numbers from a file into an array and allows the user to search the array.

The program is modular and may be adapted to support coding and testing of many algorithms that operate on arrays. The module structure of the program is:




```
/* Search.java:
   Reads file into array, searches file
   for a key.
   D. Keil 3/14
*/
import java.util.*;
import java.io.*;

public class Search
{
    public static void main(String[] args)
     throws FileNotFoundException
    {
       // Read array from disk:
       String fname = "arr1.txt";
       int[] A = readFile(fname);
       display(A);

       // Prompt for search key and
        //search for it:
       System.out.print("Search for: ");
       Scanner cin = new Scanner(System.in);
       int search_key = cin.nextInt();
       if (search (A,search_key))
          System.out.println("Found " +
              search_key);
       else
          System.out.println("Did not find "
               + search_key);
    }

static int[] readFile(String name)
 throws FileNotFoundException
// Reads from file named 'name'
 //into an array;
// returns that array.
{
   int n = numTokensInFile(name);
   int[] A = new int[n];
   FileReader reader = new
     FileReader(name);
   Scanner fin = new Scanner(reader);
   for (int i = 0; i < n; i++)
   {
       A[i] = fin.nextInt();
   }
   return A;
}

static int numTokensInFile(String name)
 throws FileNotFoundException
// See how big file named 'name' is:
{
  // Open input file:
  System.out.println("Reading "+name);
  FileReader reader =
      new FileReader(name);
  Scanner fin = new Scanner(reader);
  int n = 0;
  while (fin.hasNext())
  {
      fin.nextInt();
      ++n;
  }
  fin.close();
 return n;
}

static void display (int[] A)
// Displays elements of A
{
 for (int i = 0; i < A.length; i++)
 {
     System.out.print(A[i] + " ");
 }
 System.out.println();
}

static boolean search (int[] A, int key)
// Searches 'A' for 'key',
//returns t/f result
{
 // Traverse array looking for 'key':
 for (int i = 0; i < A.length; i++)
 {
    if (A[i] == key)
          return true;
 }
    return false;
}
}
```

Challenge: define a collection of point objects

The two Java programs below implement two separate aspects of a file-maintenance program based on a collection. They may be combined to enable you to write a program that stores data in a file, reads it into an array-based collection, and enables the user to change the content of the collection.

```
/* PointApp2.java:
   D. Keil and CS II students 3/14  */
import java.util.*;
import java.io.*;
class Point
{
  int x, y;
  Point() { x = y = 0; }  // constructor
  Point(int x0, int y0) { x = x0; y = y0; }
  public void setX(int x)
     { this.x = x; } // mutator
  public void setY(int y)
     { this.y = y; } // mutator
  public int getX() { return x; }
  public int gety() { return y; }
  public void display()
     { System.out.println("("+x+", "+y+")"); }
  public void readFile(String name)
     throws FileNotFoundException
    // Open input file, read, display a string
    {
     System.out.print("Reading "+name);
     FileReader reader = new FileReader(name);
     Scanner fin = new Scanner(reader);
     x = fin.nextInt();
     y = fin.nextInt();
     fin.close();
    }
  public void update()
    {
     // Prompt for new string:
     System.out.print("Enter new values x, y: ");
     Scanner cin = new Scanner(System.in);
     x = cin.nextInt();
     y = cin.nextInt();
    }
  public void writeFile(String name)
     throws FileNotFoundException
    // Prompt for new string, write to file
    {
     // Write new string to file:
     PrintWriter fout = new PrintWriter(name);
     System.out.print("Writing to file "+name);
     fout.println(x + " " + y);
     fout.close();
    }
}
public class PointApp2
{
    public static void main(String[] args)
     throws FileNotFoundException
    {
     Point p = new Point();
     p.readFile("Point.txt");
     p.display();
     p.update();
     p.display();
     p.writeFile("Point.txt");
    }
}
```

By adding to the program below the methods *readFile, update,* and *writeFile* defined in the previous program, you may prepare the collection class below for the addition of a user interface that supports file maintenance for the collection. That user interface could be menu based, with menu options to read from a file, insert a record, update a record, delete a record, or save to a file.

```
/* PointColxApp.java:
   D. Keil and CS II students 3/14 */
import java.util.*;
import java.io.*;

class Point
{
  int x, y;
  Point() { x = y = 0; }  // constructor
  Point(int x, int y) { this.x = x; this.y = y; }
  public void setX(int x) { this.x = x; }
  public void setY(int y) { this.y = y; }
  public int getX() { return x; }
  public int gety() { return y; }
  public void display() {
System.out.print("("+x+", "+y+") "); }
}

public class PointColxApp
{
    public static void main(String[] args)
     throws FileNotFoundException
    {
     Point[] A = new Point[4];
     A[0] = new Point(1,1);
     A[1] = new Point(1,2);
     A[2] = new Point(1,3);
     A[3] = new Point(4,4);
     A[2].setY(4);
     for (int i = 0; i < A.length; i++)
        A[i].display();
    }
}
```

Inheritance with employees and hourly employees

The program below defines a class *Employee*, with just an ID field, and a class, *Hourly*, that inherits from *Employee*, adding fields *wageRate* and *Hours*. The *main* method defines instances of each class and outputs field values, using the methods *toString* defined for each class.

The constructor of the derived class, *Hourly*, calls the constructor of *Employee*, using the keyword *super* to call the base-class constructor.

```
// HourlyApp.java
// Demonstrates inheritance with employee
classes. D. Keil,

3/14
import java.util.*;

class Employee
{
  private int ID;

  Employee() { ID = 0; }
  Employee(int id0) { ID = id0; }
  public int getID() { return ID; }
  public String toString() { return "" +
ID; }
}

class Hourly extends Employee
{
   private double wageRate,hours;

   Hourly(int id0, double w, double h)
   {
      super(id0);
      wageRate = w;
      hours = h;
   }
  public String toString()
  {
     return super.toString() + " " +
wageRate + " " + hours;
  }
}

public class HourlyApp
{
  public static void main(String[] args)
  {
     Employee emp1 = new Employee(123);
     Sys[illegible]n(emp1);
     Hou[illegible]Hourly(456,15.00,32);
     System.out.println(emp2);
  }
}
```

Base class

Simple inheritance vs. inheritance with polymorphism

Inheritance without polymorphism

The program below uses simple inheritance to demonstrate a class hierarchy, consisting of *Animal* and *Bird* classes. A bird is a kind of animal. The *main* method creates an *Animal* and a *Bird* object and calls methods that describe the motion typical of the creatures. Note that the *tellMotion* method of the *Bird* class *overrides* the *tellMotion* method of the *Animal* class, so that the *Bird* object says that birds walk and fly.

```
// Animal1.java
// Inheritance without polymorphism.
// D. Keil 3/14
import java.util.*;

class Animal
{
  Animal() { };
  public void tellMotion()
   {
    System.out.println
    ("Animals move around.");
   }
}

class Bird extends Animal
{
  Bird() { super(); };
  public void tellMotion()
   {
    System.out.println
      ("Birds walk and fly.");
   }
}

class Animal1
{
  public static void main(String[] args)
  {
    Animal a = new Animal();
    a.tellMotion();
    Bird b = new Bird();
    b.tellMotion();
  }
}
```

Output:

```
Birds walk and fly.
Animals move around.
```

Inheritance with polymorphism using an abstract base class

The program below demonstrates polymorphism using the abstract base class *Animal*. The class and two of its methods are designated *abstract*. An abstract class cannot be instantiated as objects; only its derived classes may be instantiated.

The *tellAll* method of the base class may be called by the derived *Bird* class instance. So main calls the base-class *tellAll* method, which in turn calls the derived class methods, *tellMotion* and *tellReproduction*.

```
// Animal2.java
// Inheritance with abstract base class.
// D. Keil 3/14
import java.util.*;

abstract class Animal
{
  Animal() { };
  public abstract void tellMotion();
  public abstract void tellReproduction();
  public void tellAll()
   { tellMotion(); tellReproduction(); };
}

class Bird extends Animal
{
  Bird() { super(); };
  public void tellMotion()
   {
    System.out.println
     ("Birds walk and fly ");
   }
  public void tellReproduction()
   {
    System.out.println("Birds lay eggs.");
   }
}

class Animal2
{
  public static void main(String[] args)
  {
    Bird b = new Bird();
    b.tellAll();
  }
}
```

Output:

```
Birds walk and fly.
Birds lay eggs.
```

Using bitwise operators and masks to clear and set bits

The program below uses a single 32-bit integer to store a set of integers in the range 0 to 31. Each number stored is represented as a 1 bit in a particular bit position in the 32-bit integer. Numbers are added to the set by setting a bit to 1; numbers are removed by clearing a bit to 0.

```
// num_sack.cpp
// Allows user to enter a list of numbers to
// be stored as a set in a single integer,
// then to delete numbers from the set.
// David Keil, Framingham State College, 8/98
#include <iostream.h>

typedef int bool;

void set_bit(int& num,int bitnum);
void clear_bit(int& num,int bitnum);
bool is_set(int num,int bitnum);
void display_contents(int n);

void main()
{
    int sack_of_numbers = 0,
        input;

    // Put some numbers in sack:
    do {
        cout << "Enter an integer (0..31) "
            "to put in the sack: ";
        cin >> input;
        if (input >= 0 && input <= 31)
        {
            set_bit(sack_of_numbers,input);
        }
    } while (input >= 0 && input <= 31);

    display_contents(sack_of_numbers);

    // Take some numbers back out:
    do {
        cout << "Enter an integer (0..31) "
            "to take from the sack: ";
        cin >> input;
        if (input >= 0 && input <= 31)
        {
            clear_bit(sack_of_numbers,input);
        }
    } while (input >= 0 && input <= 31);

    display_contents(sack_of_numbers);
}
```

```
void set_bit(int& num,int bitnum)
// Sets the <bitnum>th bit from right
// in <num> to 1.
{
    num |= (1 << bitnum);
}
```

Use of bitwise

```
void clear_bit(int& num,int bitnum)
// Clears to 0 the <bitnum>th bit from right
// in <num>.
{
    num &= ~(1 << bitnum);
}
```

Use of bitwise

```
bool is_set(int num,int bitnum)
// Tells whether bit <bitnum> is a 1 in
// the value <num>.
{
    int mask = 0;
    set_bit(mask,bitnum);
    return ((mask & num) != 0);
}
```

Creating a mask with many 0s and

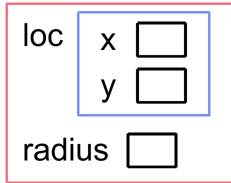
```
void display_contents(int n)
// Shows what elements are in a set
// represented by <n>.
{
    for (int i=0; i < 32; ++i)
        if (is_set(n,i))
            cout << i << " ";
    cout << endl;
}
```

Sample I/O:

```
Enter an integer (0..31) to put in the sack: 6
Enter an integer (0..31) to put in the sack: 12
Enter an integer (0..31) to put in the sack: 21
Enter an integer (0..31) to put in the sack: 2
Enter an integer (0..31) to put in the sack: 5
Enter an integer (0..31) to put in the sack: 15
Enter an integer (0..31) to put in the sack: -1
2 5 6 12 15 21
Enter an integer (0..31) to take from the sack: 12
Enter an integer (0..31) to take from the sack: 5
Enter an integer (0..31) to take from the sack: 21
Enter an integer (0..31) to take from the sack: -1
2 6 15
```

Examples of classes

The program below uses a structure type to represent circles. The circle's center is an (*x,y*) coordinate, *loc*. Both the circle and its center use structure types, so *loc* is said to be *nested*.




```
/*
	circle.c
	Initializes and displays nested-
	structure member values.
*/
#include <stdio.h>

typedef struct
{
	int x,y;
} locations;

typedef struct
{
	locations loc;
	int radius;
} circles;

void main(void)
{
	circles C = { {200,100}, 50 };
	printf("%d %d %d\n",C.loc.x,C.loc.y,
		C.radius);
}
```

Output:
20 100 50

The program *basket1.c* uses a structure type to declare basketball-team variable. *Teams* is an abstract data type in that it is defined by data attributes and an operation. The operation is a function, *display_team*, which takes a team as a parameter.

```
/*
	basket1.c
	Displays info about a college
	basketball team with most recent
	number of points scored and
	opponent's   score.
*/
#include <stdio.h>

struct Teams
{
	char name[20];
	int score,
	    opp_score;
};
typedef struct Teams teams;

void display_team(teams tm);

void main(void)
{
	teams our_team = {"Fram State",36,35};
	display_team(our_team);
}

void display_team(teams tm)
/* Displays team name, latest score, opponent score. */
{
	printf("%-20s %5d %5d\n",
	    tm.name,tm.score,tm.opp_score);
}
```

Simulating the roll of a die

Many applications, such as games and simulations, use randomization to simulate uncertainty. The program below shows some tools for generating pseudorandom numbers. The numbers are not strictly random, because the process for generating them is deterministic. The chief tools are the *stdlib.h* library and the functions *srand* and *rand* defined there.

```
/*
    rolldie.cpp
    Loops to simulate roll of a die and
    display result.
    David Keil, Framingham State College, 8/98
*/
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>

struct series_of_rolls
{
    int num_rolls;
    series_of_rolls() { num_rolls = 0; }
    void play();
};

void main(void)
{
    /* Initialize random-number generator's
        seed value, using current time: */
    time_t ltime;
    time(&ltime);
    srand(32761*ltime);

    /* Display a value returned by random-number
        generator: */
    cout << rand()
        << " is a random number in range 0 to "
        <<  INT_MAX << endl;
    series_of_rolls game;
    game.play();
}

void series_of_rolls::play()
// Prompts for length of game, simulates rolls
// of die, displays results.
{
    cout << "How many rolls of die? ";
    cin >> num_rolls;
    cout << "You rolled ";
    for (int i=0; i < num_rolls; ++i)
        cout << 1 + rand() % 6 << " ";
    cout << endl;
}
```

Sample input/output:

```
29837 is a random number in range 0 to 2147483647
How many rolls of die? 60
You rolled 6 4 3 5 3 6 6 6 3 6 6 2 6 4 2 2 3 2 4 4 1 5 2 6 4 4 2
5 3 5 3 2 6 5 6
 6 2 3 6 2 5 4 5 4 4 1 6 4 2 3 1 1 1 6 3 1 2 5 1 4
```

Testing a file-input operation

Compare the programs below to see the advantage of testing the value of a file-stream-input operation, such as *infile >> score* in the code below. Each program reads the same text file, composed of integers. This file ends with a newline character. Notice that the program on the left seems to read the value 80 twice. What happens is that the file stream reads three numbers and then attempts to read a fourth one, because the *eof* function still returns *false* since a newline remains to be read in the file.

The program on the right avoids displaying the buffer, *score*, a fourth time and incrementing *count*. It does this by testing the value of *infile >> score*. An unsuccessful attempt to input gives the extractor expression a *false* value in the *if* statement.

```
// readsco2.cpp
// Reads list of integers from file.
// May give erroneous feedback on last item.
#include<iostream.h>
#include<fstream.h>

void main()
{
    int count;
    float score;
    ifstream infile("Myscore.txt",ios::nocreate);
    if (! infile.bad())
    {
        count=0;
        while(!infile.eof())
        {
            infile >> score;
            cout << score << " ";
            ++count;
        }
        infile.close();
        cout << "count = " << count << endl;
    }
}
```

```
    Output using file with contents:
        89
        93
        80
    89 93 80 80 count = 4
```

```
// readscor.cpp
// Reads list of integers from text file.
// Handles newline at end of file correctly.
#include<iostream.h>
#include<fstream.h>

void main()
{
    int count;
    float score;
    ifstream infile("Myscore.txt",ios::nocreate);
    if (! infile.bad())
    {
        count=0;
        while(!infile.eof())
            if (infile >> score)
            {
                cout << score << " ";
                ++count;
            }
        infile.close();
        cout << "count = " << count << endl;
    }
}
```

```
    Output using file with contents
        89
        93
        80
    89 93 80 count = 3
```

Random-access file I/O example

```
/*  filesort.c
    Accepts a line of text, writes to a random-
access file,
    sorts the characters, displays sorted list. */
#include <stdio.h>
#include <conio.h>

#define LINE_SIZE 80

typedef int boolean;
const boolean FALSE=0,TRUE=1;

const char* FILE_NAME = "filesort.dat";

void create_file();
void sort_file();
void display_file();
char read_char(FILE* fp,int index);
void write_char(FILE* fp,char ch,int index);
void swap(FILE* fp,int index_1,int index_2);

void main(void)
{
   create_file();
   display_file();
   sort_file();
   display_file();
}

void create_file()
/* Accepts line of text, writes sequentially to
text file. */
{
   char line[LINE_SIZE];
   FILE* fp = fopen(FILE_NAME,"w");

   /* Prompt for text, write to file: */
   printf("Enter a line of text: ");
   gets(line);
   fputs(line,fp);
   fclose(fp);
}

void display_file()
/* Shows file contents on screen. */
{
   int i;
   FILE* fp = fopen(FILE_NAME,"r");
   char input;

   printf("Contents of file %s, sorted:
",FILE_NAME);
   while((input = fgetc(fp,i)) != EOF)
      putchar(input);
   fclose(fp);
}

void sort_file()
/* Uses Bubble algorithm to put characters in file
in order. */
{
   boolean did_swap;
   int i;
   FILE* fp = fopen(FILE_NAME,"rb+");
   long int file_length;

   /* Find length of file: */
   fseek(fp,0,SEEK_END);
   file_length = ftell(fp);

   /* Sort it in place: */
   printf("\rSorting\n");
   do  {
      did_swap = FALSE;
      for (i=0; i < file_length-1; ++i)
         if (read_char(fp,i) > read_char(fp,i+1))
         {
            swap(fp,i,i+1); // swap values in
file
            did_swap = TRUE;
         }
   }
   while (did_swap);
   fclose(fp);
}

char read_char(FILE* fp,int index)
/* Reads, returns the <index>th character in the
random file. */
{
   char input;

   fseek(fp,index,SEEK_SET);
   input = fgetc(fp);
   return input;
}

void write_char(FILE* fp,char ch,int index)
/* Writes <ch> to the <index>th location of the
random file. */
{
   fseek(fp,index,SEEK_SET);
   fputc(ch,fp);
}

void swap(FILE* fp,int index_1,int index_2)
/* Swaps the values at locations <index_1>,
<index_2> of random file. */
{
   // Get values in file into variables:
   char old_1 = read_char(fp,index_1),
        old_2 = read_char(fp,index_2);

   // Copy from <index_2> to <index_1>:
   write_char(fp,old_2,index_1);

   // Store old value from <index_1> at <index_2>:
   write_char(fp,old_1,index_2);
}
```

Member functions to display, retrieve, and save one rational-number object

```
/*
   rationl2.cpp
   Reads 2 rational-number objects from disk,
   saves result to disk.

   David Keil, Framingham State College, 1/99
*/
#include <iostream.h>
#include <fstream.h>

class rationals
{
public:
   rationals()
      { numerator = denominator = 0; }
   rationals(int n,int d)
      { numerator = n; denominator = d; }
   rationals plus(rationals term)
      {  return rationals(numerator * term.denominator
+
                 denominator * term.numerator,
                 denominator * term.denominator); }
   void display()
      { cout << numerator << " / " << denominator <<
endl; }
   void retrieve(ifstream& fin)
      { fin >> numerator >> denominator; }
   void save(ofstream& fout)
      { fout << numerator << " " << denominator; }
private:
   int numerator,denominator;
};

void main()
{
   // File input:
   ifstream fin("rationl2.in",ios::nocreate);
   rationals input1,input2;
   input1.retrieve(fin);
   input2.retrieve(fin);
   input1.display();
   input2.display();
   fin.close();
   // Calculation:
   rationals result = input1.plus(input2);
   // File output:
   result.display();
   ofstream fout("rationl2.out");
   result.save(fout);
   fout.close();
}
```

Given these contents of *rationl2.in*:

```
2 3
1 6
```

the program has this screen output:

```
2 / 3
1 / 6
15 / 18
```

and writes this to *rationl2.out*:

```
15 / 18
```

Pseudocode for collections with file I/O

Item class interface

User input

Prompt for each member data item

Screen display

Show each member data item, formatted (without header)

File input / read from file (with *ifstream* parameter)

Read member data from file

File output / save to file (with *ofstream* parameter)

Write member data to file

Collection class interface

User input

Loop through array, calling item user input function

Screen display

Display header

Loop through array, calling item screen display function

File input / file retrieve

Open file for input with *ifstream*

Loop through array, calling item file-input function with *ifstream* parameter

Close file

File output / file save

Open file for output with *ofstream*

Loop through array, calling item file-output function with *ofstream* parameter

Close file

main

1. Declare collection object using collection class
2. If file exists read and display collection from file
3. Update collection if desired, using collection-input function
4. Call collection display function
5. If updated write collection back to file with save function

A collection of points

The program at the right shows how to build a collection using a class to describe the objects that are to comprise the collection (points on a coordinate axis here) and a class to encapsulate the collection, using an array and an integer. The integer records the current size of the collection.

The program reads and displays the file *point.txt*, whose text is as follows:

```
3 9
0 0
2 4
1 1
4 16
```

The screen output is formatted by adding parentheses and commas.

```
// point.cpp
// Implements collection of (x,y)-coordinate
// points, using array of objects.
// Reads a series of points from file,
// displays them.
#include <bool.h>
#include <iostream.h>
#include <fstream.h>

class points
{
public:
    points() { x = y = 0; }
    points(int x_init,int y_init)
        { x = x_init; y = y_init; }
    bool input(istream& in)
        { return !!(in >> x >> y); }
    void display()
        { cout << "(" << x << "," << y << ") "; }
private:
    int x,y;
};

class point_collections
{
public:
    point_collections()
        { num_points = 0; }
    void retrieve(char* file_name);
    void display();
    enum { MAX_POINTS = 100 };
private:
    points point[MAX_POINTS];
    int num_points;
};

void main()
{
    point_collections list;
    list.retrieve("point.txt");
    list.display();
}

// Class <point_collections>:

void point_collections::retrieve(char* file_name)
// Reads from file <file_name> into collection.
{
    ifstream fin(file_name,ios::nocreate);
    while(point[num_points].input(fin))
        num_points++;
    fin.close();
}

void point_collections::display()
// Outputs all items to screen.
{
    for (int i=0; i < num_points; ++i)
        point[i].display();
}
```

Output:

```
(3,9) (0,0) (2,4) (1,1) (4,16)
```

An array of objects: a basketball league

```
/*
        basketba.cpp
        Displays a list of college basketball teams with
        most recent number of points scored and opponent's score.
*/
#include <iostream.h>
#include <iomanip.h>

const int MAX_TEAMS = 50;

struct teams
{
        char name[20];
        int score,
            opp_score;
        void display();
};


void main()
{
        teams team[] = {
            {"Babson",66,58},{"BC",84,59},{"Emerson",35,36},
            {"Fram State",36,35},{"MIT",54,51},{"Mt Holyoke",51,54},
            {"Pine Manor",56,72},{"Pittsburgh",59,84},{"Tufts",72,56},
            {"UConn",67,54},{"Villanova",54,67},{"WPI",58,66}
            };
        int num_teams = sizeof(team) / sizeof(teams);
        for (int i=0; i < num_teams; i++)
            team[i].display();
}

void teams::display()
{
        cout << setw(20) << name << setw(5) << score
                 << setw(5) << opp_score << endl;
}
```

Class to encapsulate sports teams

Program output:

```
   Emerson   35   36
Fram State   36   35
       MIT   54   51
Mt Holyoke   51   54
Pine Manor   56   72
Pittsburgh   59   84
     Tufts   72   56
     UConn   67   54
 Villanova   54   67
       WPI   58   66
```

Linking separately compiled files into a project

The five files below form a *project*, consisting of a source file that defines a *main* function and a set of other source files that declare constants, types and functions, and other source files that define functions. The file containing *main* (*pointdrv.cpp*, here) is called a driver. The files that declare identifiers have the extension *.h* and are called *headers*. The headers here define class *points* and *pointclx*.

In Microsoft Visual C++, Version 4.0, you create a project by pressing *File / New / Project Workspace*. Give the project the same name (minus *.cpp*) as your source file containing *main*. Then choose *Insert / Files Into Project* for each *.cpp* file that belongs in the project.

When you give the *Build* command with the project workspace open, the IDE will link all *.cpp* files together so that the functions defined in each one will be available to all the others.

```
/*
    pointdrv.cpp
    Implements collection of (x,y)-coordinate
    points, using array of objects.
    Reads a series of points from file,
    displays them.
    Link with points.cpp, pointclx.cpp

    David Keil, Framingham State College, 1/99
*/
#include "pointclx.h"

void main()
{
    point_collections list;
    list.retrieve("point.txt");
    list.display();
}
```

```
// points.h
// Class declaration for points.
#include <iostream.h>
#include <fstream.h>

#define bool int

class points
{
public:
    points();
    points(int x_init,int y_init);
    bool input(istream& in);
    void display();
private:
    int x,y;
};
```

```
// points.cpp
// Class <points>
#include "points.h"

points::points()
{
    x = y = 0;
}

points::points(int x_init,int y_init)
{
    x = x_init; y = y_init;
}

bool points::input(istream& in)
{
    return !!(in >> x >> y);
}

void points::display()
{
    cout << "(" << x << "," << y << ") ";
}
```

```
// pointclx.h

#include "points.h"

class point_collections
{
public:
    point_collections();
    void retrieve(char* file_name);
    void display();
    enum { MAX_POINTS = 100 };
private:
    points point[MAX_POINTS];
    int num_points;

}
```

```
// pointclx.cpp
// Class <point_collections>:
#include "pointclx.h"

point_collections::point_collections()
{
    num_points = 0;
}

void point_collections::retrieve(char* file_name)
// Reads from file <file_name> into collection.
{
    ifstream fin(file_name,ios::nocreate);
    while(point[num_points].input(fin))
        num_points++;
    fin.close();
}

void point_collections::display()
// Outputs all items to screen.
{
    for (int i=0; i < num_points; ++i)
        point[i].display();
}
```

Examples with pointers and dynamic allocation

```
/*
	ptrdemo.c
	Declares int variable and a pointer to
	it, shows values assigned and a
	retrieved by dereferencing.

*/
#include <stdio.h>

void main(void)
{
	int n = 5;
	int *p;  /* pointer variable */

	p = &n;  /* assign value to pointer */
	printf("*p = %d\n",*p);
	n = 9;
	printf("*p = %d\n",*p);
	*p = 2;  /* dereference pointer */
	printf("n = %d\n",n);
}
```

```
/*
	intaddr.c
	Displays address of a variable.
*/
#include <stdio.h>

int global;

void main(void)
{
	int local;

	printf("&global = %i\n",&global);
	printf("&local  = %i\n",&local);
}
```

```
/*	tally.c:
	Prompts for up to five integers,
	stores as array, displays array. */
#include <stdio.h>

typedef int BOOLEAN;

#define MAX_SCORES 5

/* Define integer-collection structure type
	and associated functions: */
struct tally_tag
{
	int score[MAX_SCORES],
	    num_scores;
};
typedef struct tally_tag tallies;

BOOLEAN tally_input(tallies* pt);
void tally_display(tallies t);

void main(void)
{
	tallies tally;

	if (tally_input(&tally))
		tally_display(tally);
}

/* Function definitions for <tallies> ADT: */

BOOLEAN tally_input(tallies* pt)
/* Inputs list of integers from keyboard,
	assigns it to elements of array in *pt. */
{
	int input;

	/* Input elements: */
	pt->num_scores = 0;
	do {
		printf("Enter a score (0 to quit): ");
		scanf("%d",&input);
		if (input != 0)
		{
			pt->score[pt->num_scores] = input;
			pt->num_scores++;
		}
	} while (pt->num_scores < MAX_SCORES &&
			input != 0);
	return (pt->num_scores > 0);
}

void tally_display(tallies t)
/* Display the array in <t>. */
{
	int i;

	for (i=0; i < t.num_scores; ++i)
		printf("%d ",t.score[i]);
	printf("\n");
}
```

```
/*
	listing.c
	Dynamically allocates telephone-listing
	structure, assigns values to members.
*/
#include <stdio.h>

typedef struct
{
	char name[40];
	int extension;
} listings;

void main(void)
{
	listings* p_listing;

	p_listing = (listings*)
		malloc(sizeof(listings));
	strcpy(p_listing->name,"Charlie Brown");
	p_listing->extension = 242;
	printf("%s %d\n",
		p_listing->name,p_listing->extension);
}
```

```
/*
	listing2.c
	Dynamically allocates telephone-listing
	structure, assigns values to members.
*/
#include <stdio.h>

typedef struct
{
	char name[40];
	int extension;
} listings;

listings* listing_init(char *nm,int ext);

void main(void)
{
	listings* p_listing =
		listing_init("Jane Chen",120);

	printf("%s %d\n",
		p_listing->name,p_listing->extension);
}

listings* listing_init(char *nm,int ext)
/* Allocates and initializes <listings>
dynamic variable, returns pointer to it. */
{
	listings* p_listing = (listings*)
		malloc(sizeof(listings));
	strcpy(p_listing->name,nm);
	p_listing->extension = ext;
	return p_listing;
}
```

```
/*
	roster.c
	Accepts an array of dynamically
	allocated strings.
*/
#include <stdio.h>
#define MAX_NAMES 100

typedef struct
{
  char *name[MAX_NAMES];
  int size;
} rosters;

void main(void)
{
	rosters roster;
	char input[80];

	roster.size = 0;
	do
	{
		printf("Name? ");
		gets(input);
		if (input[0] != '\0')
		{
			roster.name[roster.size] =
				(char*)malloc(sizeof input + 1);
			strcpy(roster.name[roster.size],
				input);
			++roster.size;
		}
	}
	while (roster.size < MAX_NAMES &&
				input[0] != '\0');
	printf("%d names entered.\n",roster.size);
}
```

A single phone-book listing on the heap

```
/* listing.cpp
  Dynamically allocates phone-listing
  structure, assigns values to members. */
#include <iostream.h>
#include <string.h>

struct listings
{
    char name[40];
    int extension;
};

void main()
{
  listings* p_emp = new listings;
  if (p_emp != NULL)
  {
    strcpy(p_emp->name,"Charlie Brown");
    p_emp->extension = 242;
    cout << p_emp->name << p_emp->extension << endl;
  }
  else
    cout << "Out of memory\n";
}
```

A collection of employees

The program below shows how to declare a collection class to represent a roster of employee objects, using dynamic allocation of an array of these objects. Two classes are used: *employees*, with a name and a wage member; and *rosters*, with a pointer-to-*employees* member and an integer counter that keeps track of the number of employees in the collection.

When a roster is declared in *main*, an array of employee objects is created dynamically on the heap. When an employee is inserted into the collection, the member data for the employee is stored in one element of that dynamically allocated array.

```
// roster.cpp
// Uses classes to define an array of
// employee records, assigns value to one,
// displays it.
#include <iostream.h>
#include <string.h>
#include <iomanip.h>

const int MAX_EMPLOYEES = 50;
class employees
{
        char name[40];
        float wage;
public:
        employees()
              { *name = '\0'; wage = 0; };
        employees(char *nm,float wg)
              { strcpy(name,nm); wage = wg; };
        void set(char *nm,float wg)
              { strcpy(name,nm); wage = wg; };
        void display()
              { cout << setw(15) << name
                    << setiosflags(ios::showpoint|
                          ios::fixed)
                    << setprecision(2) << setw(8)
                    << wage << endl; };
};

class rosters
{
        employees *emp;
        int max_size,curr_size;
public:
        rosters() { curr_size = 0; };
        rosters(int sz);
        ~rosters() { delete[] emp; };
        void insert(char *nm,float wg);
        void display();
};

void main()
{
        rosters payroll(MAX_EMPLOYEES);
        payroll.insert("Jones",15.00);
        payroll.insert("Kang",15.25);
        payroll.display();
}

// Class <rosters>:

rosters::rosters(int sz)
// Constructor that allocates memory to
// employee array.
{
        max_size = sz;
        emp = new employees[max_size];
        curr_size = 0;
}

void rosters::insert(char *nm,float wg)
// Adds object to array.
{
        emp[curr_size].set(nm,wg);
        ++curr_size;
}

void rosters::display()
// Displays each element of array.
{
        for (int i=0; i < curr_size; ++i)
              emp[i].display();
}
```

A class library for safe string manipulation

The *interface* of a class is its set of member-function prototypes. *Client code*, written by the application programmer, declares instances of the class and has these instances, or objects, call *member functions* of the class.

The ANSI/ISO (American National Standards Institute / International Standards Organization) draft standard for C++ includes the specification for a class called *string* that provides a convenient and safe standard data type for character strings. In the C++ standard, unlike with C-style strings, a string is an object, with member functions and associated operators, not an array. Its identifier is not a pointer and operations on strings are checked automatically for validity. The ANSI/ISO standard string class is available if you write *#include <string>* in your program. It is supported by Microsoft Visual C++ Version 5.0.

In this course we will use a class called *strings*, similar to the draft standard *string*. It is declared in the header file *strings.h* and its member functions are defined in the file *strings.cpp*, available at *m: \ Resources \ Faculty \ dkeil \ 63259 \ 01 stream*

An instance of *strings* looks like this in memory:

content [| | | | | | | | | | | | | | | | | |] ...

Part of the interface of class *strings*:

`strings::strings()`
Default constructor
Example: `strings s1;`

`strings::strings(char* s)`
Conversion constructor from *char**.
Example: `strings s1("string1");`

`boolean strings::assign(char* s)`
Copies *s* content to *content*
Example: `s1.assign("hello");`

`boolean strings::append(char* s)`
Like *strcat*.
Example: `s1.append("ing");`

`boolean strings::append(strings s)`
Like *strcat*.
Example: `s1.append(s2);`

`int strings::compare(char* s)`
Like *strcmp*.
Example: `s1.compare(s2);`

`unsigned int strings::length()`
Like *strlen*.
Example: `int len = s1.length();`

`char strings::at(unsigned int n)`
Returns *n*th character in *content*.
Example: `char ch = s1.at(3);`

`char strings::set_at(unsigned int n,char ch)`
Assigns *ch* as *n*th element of *content*.
Example: `s1.set_at(0,'P');`

`boolean strings::insert(unsigned int loc, char* s)`
If *s* is not too long, inserts *s* into *content* at location *loc*.
Example: `s1,insert(2,"XXX");`

`boolean strings::delet(unsigned int loc, int num_chars)`
Deletes *num_chars* from *content*, starting at location *loc*.
Example: `s1.delet(2,4);`

`char* strings::substr(unsigned int loc,int num_chars)`
Returns pointer to a static string containing substring of *content* starting at *loc* that contains *num_chars* characters.
Example: `cout << s1.substr(3,2);`

`boolean strings::replace(unsigned int loc,char* s)`
Replaces text in *content* at location *loc* with contents of *s*.
Example: `s1.replace(2,"QQQ");`

`void strings::input(istream& is)`
Stream input.
Example: `s1.input(cin);`

`void strings::display(ostream& os)`
Stream output.
Example: `s1.display(cout);`

Example of client code that uses *strings*

```
// strgdemo.cpp
// Demonstrates safe-string library
// class <strings>.
#include "strings.cpp"

void main()
{
   // Declare, initialize, display two strings:
   // Declare, initialize, display two strings:
   strings string1,
           string2("string2");
   string1.assign("string1");
   string1.display(cout);
   cout << endl;
   string2.display(cout);
   cout << endl;

   // Get two from user:
   cout << "Enter a string: ";
   string1.input(cin);
   cout << "Another: ";
   string2.input(cin);
   string1.append(string2);
   cout << "Length of your strings together is "
       << string1.length() << ".\n";
   string1.display(cout);
   cout << endl;
}
```

Sample input/output:

```
string1
string2
Enter a string: Howdy
Another: Bye
The length of your strings together is 8.
HowdyBye
```

Overloaded operators provide an even more convenient interface for string classes

In the C language, all operators, such as =, +, <, are as defined by the C compiler. But in C++, we may also extend, or overload, these operators to apply to instances of a class that we define. The ANSI/ISO *string* class and the instructor's predefined *strings* class provide a number of overloaded operators for use with string objects, so that they application programmer may work with strings in a similar way to how the standard types are manipulated. For example, the overloaded << operator for *strings* objects makes this sequence of statements possible:

```
strings greeting = "Hello";
cout << greeting << endl;
```

so that "Hello" is displayed.

The program at left could be coded in the following way:

```
strings string1,
      string2("string2");
string1 = "string1";
cout << string1 << endl << string2 << endl;

// Get two from user:
cout << "Enter a string: ";
cin >> string1;
cout << "Another: ";
cin >> string2;
string1 += string2;
cout << "Length of your strings together is "
    << string1.length() << endl << string1
    << endl;
```

The following operators are defined for class *strings*:

Operator	Left operand	Right operand	Operation	Example (given *strings s;*)
<<	output stream	*strings*	stream output	*cout << s;*
>>	input stream	*strings*	stream input	*cin >> s;*

The following operators take a *strings* object as their left operands:

Operator	Right operand	Operation	Example (given *strings s1,s2;*)
=	*strings; char*; char*	assignment	*s1 = "schoolbus";;*
[ ]	*int*	access to element[subscript]	*char c = s1[0];*
( )	two *int*s (start, length)	substring	*cout << s1(0,6);*
>, >=, <, <=, ==, !=	*strings; char*; char*	string comparison	*if (s1 < s2) cout << s1 << "comes first";*
+	*strings; char*; char*	concatenation	*cout << s1 + s2;*
+=	*strings; char*; char*	concatenation to self	*s1 += s2;*

A set of operators similar to the set shown above is defined for the ANSI/ISO *string* class.

A program to manage employee objects

```
// employee.cpp
// Initializes and displays
// employee objects.
#include <iostream.h>
#include <string.h>
#include <iomanip.h>

const int NAME_LEN = 30;

class employees
{
public:
   employees();
   employees(char* nm,int I,float sal);
   char* get_name() const;
   void set_name(char* nm);
   void display() const;
private:
   char name[NAME_LEN];
   int ID;
   float salary;
};

void main()
{
   employees emp1("Jones",1078,48000.0),
             emp2("Smith",2392,61034.0),
             emp3;
   emp1.display();
   emp2.display();
   cout << "Emp2's name is "
      << emp2.get_name() << endl;
   emp2.set_name("Smythe");
   emp2.display();
}

employees::employees()
// Default constructor.
{
   name[0] = ID = salary = 0;
}

employees::employees(char* nm,int I,
        float sal)
// Constructor.
{
   strcpy(name,nm);
   ID = I;
   salary = sal;
}

void employees::display()
// Shows all member values.
{
   cout << setiosflags
      (ios::left | ios::showpoint)
      << setprecision(2)
      << setw(20) << name
      << setiosflags(ios::right | ios::fixed)
      << setw(8) << ID
      << setw(12) << setprecision(2)
      << salary << endl;
}

char* employees::get_name()
// Selector access function.
{
   return name;
}

void employees::set_name(char* nm)
// Mutator access function.
{
   if (strlen(nm) < NAME_LEN)
      strcpy(name,nm);
}
```

Output:

```
Jones                1078    48000.00
Smith                2392    61034.00
Emp2's name is Smith
Smythe               2392    61034.00
```

A program to manage a collection of integers

The program below stores an array safely inside a class. It checks before appending a new element to the array elements, to see if the array is already full. Similar bounds checks could be implemented to filter any attempt to access an array element.

```
// score.cpp
// Prompts for and displays a series of
// test or game scores.
#include <iostream.h>

typedef int bool;
const bool FALSE = 0, TRUE = 1;

// A score list is a collection of integers:
class score_lists
{
   enum {MAX_SCORES = 6};
public:
   score_lists() { size = 0; };
   bool append(int new_element);
   void input();
   void display();
private:
   int element[MAX_SCORES];
   int size;
};

void main()
{
   score_lists list;
   list.input();
   list.display();
}

void score_lists::input()
// Keyboard-input member function.
{
   int input_value;
   do {
      cout << "Enter score (-1 to quit): ";
      cin >> input_value;
   } while (input_value >= 0 &&
              append(input_value));
}

bool score_lists::append(int new_element)
// Adds <new_element> to collection.
{
   if (size < MAX_SCORES)
   {
      element[size++] = new_element;
      return TRUE;
   }
   else
   {
      cout << "score_lists::append: "
         "collection already full" << endl;
      return FALSE;
   }
}

void score_lists::display()
// Displays values.
{
   cout << endl;
   for (int i = 0; i < size; ++i)
      cout << element[i] << endl;
}
```

Sample I/O (2 runs):

```
Enter score (-1 to quit): 90
Enter score (-1 to quit): 42
Enter score (-1 to quit): 64
Enter score (-1 to quit): -1
90
42
64
Enter score (-1 to quit): 90
Enter score (-1 to quit): 86
Enter score (-1 to quit): 98
Enter score (-1 to quit): 75
Enter score (-1 to quit): 92
Enter score (-1 to quit): 84
Enter score (-1 to quit): 95
score_lists::append: collection already full
Enter score (-1 to quit): -1
90
86
98
75
92
84
95
```

A rational-numbers class with the addition operator

The following program defines a class to implement rational numbers, each having a numerator and a denominator. The + operator, ordinarily not defined for structure types or classes, is implemented. The class has a friend inserter operator as well.

```
// ration1.cpp
// Initializes, adds 2 rational numbers.
#include <iostream.h>

class rationals
{
   int numerator,
       denominator;
public:
   rationals(int num,int den):
      numerator(num),denominator(den) {}
   rationals operator+(rationals term2);
   friend ostream& operator<<
      (ostream& os,rationals r);
};

rationals rationals::operator+(rationals term2)
// Returns rational object expressing sum of
// calling object and <term2>.
{
   int sum_numerator =
      numerator * term2.denominator +
      denominator * term2.numerator;
   int sum_denominator =
      denominator * term2.denominator;
   rationals sum(sum_numerator,sum_denominator);
   return sum;
}

ostream& operator<<(ostream& os,rationals r)
// Overloaded inserter.
{
   cout << r.numerator << "/" << r.denominator;
   return os;
}

void main()
{
   // Add one-half and two-
thirds:
   rationals a(1,2),b(2,3);
   rationals sum = a + b;
   cout << a << " + " << b
        << " = " << sum << endl;
}
```

Adds two rational-number objects

Output:

```
1/2 + 2/3 = 7/6
```

Overloading relational operator for a class of employees

The program below defines a class, *parts*, that has an overloaded operator, >, as a member function. The expression in *main*, (part1 > part2), uses this operator function.

The program also declares a global operator function, <<, that outputs a *parts* instance to a stream. This function is a friend of *parts* because it uses data members of a *parts* object.

```
// employe2.cpp
// Initializes and displays an employee object,
// using some overloaded operators.
#include <iostream.h>
#include <string.h>

typedef int bool;

class employees
{
public:
   employees(char* n,int i,double sal):
     ID(i),salary(sal) { strcpy(name,n); }
   char* get_name() { return name; }
   bool operator<(employees emp2);
private:
   char name[40];
   int ID;
   double salary;
};

bool employees::operator<(employees emp2)
// Less-than relational operator.
{
   return (strcmp(name,emp2.get_name()) < 0);
}

void main()
{
   employees emp1("Smith",1078,48000.0),
             emp2("Jones",2392,61034.0);
   if (emp1 < emp2)
      cout << emp1.get_name();
   else
      cout << emp2.get_name();
   cout << " comes first" << endl;
}
```

Compares strings stored as name

Output:

```
Jones comes first
```

A file-maintenance program for a collection of employee objects

The program below, in five source files, implements a collection of objects of class *employees* using an array of employees. Class *employees* is defined in files *employee.h* and *employee.cpp*. The collection class is *payrolls*, defined in *payroll.h* and *payroll.cpp.*

Alternative implementations of *payrolls* could use the same *employees* source files. Implementations in the example code subdirectory *04 Containers* include an array of pointers to employees (*payroll2.h, payroll2.cpp, roster2.cpp*), a dynamically allocated array of employees (*payroll3.h, payroll3.cpp, roster3.cpp*), a linked list (*emplist.cpp*), and a stack (*empstack.cpp*).

```
// payroll.h
// Declares container class <payrolls>,
// organized as array of employee objects.
// David Keil, Framingham State College, 6/98
#ifndef PAYROLL_H
#define PAYROLL_H

#include "employee.h"

const int MAX_RECS = 100;

class payrolls
{
public:
   payrolls();
   payrolls(char nm[]);
   ~payrolls();
   void display();
   void add_rec();
   void delete_rec();
   void edit_rec();
   bool retrieve();
   void save();
private:
   char file_name[40];
   employees emp[MAX_RECS];
   int num_recs;
};

#endif // #ifndef PAYROLL_H
```

```
// roster.cpp
// Maintains a payroll roster as a collection
// of employee objects. Options: save to disk,
// retrieve, edit, insert, delete employee.
#include "payroll.cpp"

int run_menu(char *option, ...);

void main()
{
   enum options {opQuit,opAdd,opDelete,opEdit};
   payrolls roster("payroll.dat");
   if (roster.retrieve())
   {
      int option;
      do {
         roster.display();
         cout << "\n\nChoose an operation: \n";
         option = run_menu("Quit","Add rec",
            "Delete rec","Edit rec","");
         switch (option)
         {
            case opAdd:
               roster.add_rec();   break;
            case opDelete:
               roster.delete_rec(); break;
            case opEdit:
               roster.edit_rec();   break;
         }
      } while (option != opQuit);
   }
   // Destructor automatically prompts to save
   // data as <roster> goes out of scope.
}
// Global functions:
int run_menu(char *option, ...)
// Displays a series of <option> arguments,
// gets integer from user, returns ordinal value
// of option chosen. Last argument must be "".
{
   // Display variable number of arguments:
   va_list ap;   // declared in <stdarg.h>
   va_start(ap,option);
   for (int num_args=0; *option; ++num_args)
   {
      cout << num_args << ". "
         << option << endl;
      option = va_arg(ap,char *);
   }
   va_end(ap);
   // Get user choice:
   int retval;
   cin >> retval;
   if (retval >= 0 && retval <= num_args)
      return retval;
   else
      return 0;
}
```

```
// payroll.cpp
// Member function definitions for class
// <payrolls>, employee list organized as
// array of objects.
// David Keil, Framingham State College, 6/98
#include "payroll.h"
#include "employee.cpp"

payrolls::payrolls(char nm[])
// Constructor.
{
   num_recs = 0;
   strcpy(file_name,nm);
}

payrolls::~payrolls()
// Destructor. Prompts to save all records.
{
   cout << "Save file? ";
   char ch;
   cin >> ch;
   if (ch == 'y')
      save();
}
void payrolls::edit_rec()
// Prompts for record #, new value.
{
   cout << "Record # to edit: ";
   int rec_num;
   cin >> rec_num;
   if (rec_num > 0 && rec_num <= num_recs)
   {
      cout << "Record: ";
      emp[rec_num-1].display();
      cout << ".  New value: ";
      emp[rec_num-1].input_from_user();
   }
}

bool payrolls::retrieve()
// Retrieves data file from disk into array.
{
   ifstream infile(file_name,ios::nocreate);
   if (infile.fail())
   {
      cout << file_name << " not found\n";
      return FALSE;
   }
   else
   {
      while (emp[num_recs].retrieve(infile)
            && num_recs < MAX_RECS)
         ++num_recs;
      infile.close();
      return TRUE;
   }
}
void payrolls::save()
// Prompts user for permission to save data
// to disk, does so.
{
   ofstream outfile("payroll.dat");
   for (int i=0; i < num_recs; ++i)
      emp[i].save(outfile);
   outfile.close();
}

void payrolls::display()
// Presents array on screen with items numbered.
{
   cout << "\nFile " << file_name << ":\n";
   for (int i=0; i < num_recs; ++i)
   {
      cout << i + 1 << ". ";
      emp[i].display();
   }
}

void payrolls::add_rec()
// Prompts for part name, appends it to
// array of records.
{
   if (num_recs < MAX_RECS)
      emp[num_recs++].input_from_user();
}

void payrolls::delete_rec()
// Prompts for record number, deletes record.
{
   cout << "# of record to delete: ";
   int rec_num;
   cin >> rec_num;
   if (rec_num > 0 && rec_num <= num_recs)
   {
      for (int i=rec_num-1; i <= num_recs; ++i)
         emp[i] = emp[i+1];
      --num_recs;
   }
}
```

```
// employee.h
// Declares a class to encapsulate an employee.
// David Keil, Framingham State College, 6/98
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <iostream.h>
#include <string.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdarg.h>

typedef int bool;
const bool FALSE = 0,TRUE = 1;

class employees
{
   char ID[12];
   char name[80];
   long int salary;
public:
   employees();
   employees(char* id,char* nm,long int sal);
   void display();
   void input_from_user();
   bool retrieve(ifstream& is);
   void save(ofstream& os);
   friend ostream& operator<<
      (ostream& os,employees e);
};

#endif // define EMPLOYEE_H
```

```
// employee.cpp
// Defines member functions of class
// <employees>.
// David Keil, Framingham State College, 6/98
#include "employee.h"

employees::employees()
// Default constructor.
{
   *ID = *name = 0;
   salary = 0;
}

employees::employees(char* id,char* nm,
   long int sal)
// Constructor.
{
   strcpy(ID,id);
   strcpy(name,nm);
   salary = sal;
}

void employees::display()
// Shows all member values.
{
   cout << setiosflags(ios::left)
       << setw(12) << ID
       << setw(20) << name
       << setiosflags(ios::right)
       << setw(8) << salary << endl;
}

void employees::input_from_user()
// Prompts for all member values.
{
   cout << "Employee ID, last name, salary: ";
   cin >> ID >> name >> salary;
}

bool employees::retrieve(ifstream& is)
// Reads member values from file <is>, if not
// end of file.
{
   if (is.bad() || is.eof())
      return FALSE;
   else
   {
      is >> ID >> name >> salary;
      return (! is.eof());
   }
}

void employees::save(ofstream& os)
// Saves member values to file <os>.
{
   os << ID << name << salary << endl;
}

ostream& operator<<(ostream& os,employees e)
// Inserter.
{
   cout << setw(8) << e.ID << setw(20) << e.name
       << setw(8) << e.salary << endl;
   return os;
}
```

Defining and using an iterator class

```
// scoriter.cpp
// Defines and inputs collection of scores,
// uses iterator to display it.
// David Keil, Framingham State College, 10/98
#include <iostream.h>

typedef int bool;
const bool false=0,true=1;

class int_collections
{
public:
   int_collections() { num_scores = 0; };
   void input();
   void display();
   friend class IC_iterators;
private:
   enum {MAX_INTS = 50};
   int element[MAX_INTS];
   int num_scores;
};

class IC_iterators
{
public:
   IC_iterators(int_collections* p_arg)
      { p_collection = p_arg; };
   void first() { current = 0; }
   bool next();
   bool operator!=(int subscript);
   int operator*();
private:
   int_collections* p_collection;
   int current;
};

// Class <int_collections>:

void int_collections::input()
// Prompts for scores until user enters negative
value.
{
   int input;
   for( ; ; )
   {
      cout << "Enter an integer (-1 to quit): ";
      cin >> input;
      if (input >= 0 && num_scores < MAX_INTS)
         element[num_scores++] = input;
      else
         break;
   }
}

// Class <IC_iterators>:

bool IC_iterators::next()
{
   int ok = (current < int_collections::MAX_INTS-1);
   if (ok)
      ++current;
   return ok;
}

bool IC_iterators::operator!=(int subscript)
{
   if (subscript == NULL)
      return (current < p_collection->num_scores);
   else
      return (current != subscript);
}

int IC_iterators::operator*()
{
   return p_collection->element[current];
}

// MAIN

void main()
{
   int_collections score_history;
   IC_iterators I(&score_history);

   // Apply input and output routines to
   // each item in the collection:
   score_history.input();
   for (I.first(); I != NULL; I.next())
      cout << *I << " ";
   cout << endl;
}
```

Sample I/O:

```
Enter an integer (-1 to quit): 3
Enter an integer (-1 to quit): 5
Enter an integer (-1 to quit): 7
Enter an integer (-1 to quit): 9
Enter an integer (-1 to quit): -1
3 5 7 9
```

Linked list, stack, and binary search tree classes

The three programs below show how some classic data structures are implemented and how they support the abstract concept of a collection.

```
// emplist.cpp
// Builds a linked list of employees from file,
// displays it.
#include <iostream.h>
#include <string.h>
#include <ctype.h>
#include "employee.cpp"

class list_nodes
{
public:
   list_nodes() { next = NULL; };
   list_nodes(employees e)
      { emp = e; next = NULL; };
   employees get_value() const { return emp; };
   list_nodes *get_next() { return next; };
   friend class employee_lists;
private:
   employees emp;
   list_nodes *next;
};

class employee_lists
{
public:
   employee_lists() { header.next = NULL; };
   bool retrieve(char* file_name);
   void prepend(employees e);
   void display();
   list_nodes *first() { return header.next; };
   list_nodes *get_header() { return &header; };
private:
   list_nodes header;
};

bool employee_lists::retrieve(char* file_name)
// Reads list of employees from file.
{
   ifstream infile(file_name,ios::nocreate);
   if (infile.fail())
   {
      cout << file_name << " not found\n";
      return FALSE;
   }
   else
   {
      employees input;
      while (input.retrieve(infile))
         prepend(input);
      infile.close();
      return TRUE;
   }
}
void employee_lists::prepend(employees e)
// Inserts a node containing data value <s> at
// the beginning of a linked list.
{
   list_nodes *new_node = new list_nodes(e);
   if (header.next == NULL)
      header.next = new_node;
   else
   {
      new_node->next = header.next;
      header.next = new_node;
   }
}

void employee_lists::display()
// Outputs data values of all nodes of list.
{
   list_nodes *p_node = header.next;
   while (p_node != NULL)
   {
      p_node->emp.display();
      p_node = p_node->next;
   }
}

void main()
{
   // Build list from user input, and display:
   employee_lists list;
   list.retrieve("payroll.dat");
   list.display();
}
```

The following program shows how simple a collection may be to build if a class is already defined for the contained object (*employees*) and if a class template for the data structure (stacks here) is defined.

```
// empstack.cpp
// Builds a linked list of employees from file,
// displays it.
#include "employee.cpp"
#include "liststac.h"

void main()
{
   // Build list from user input, and display:
   stacks<employees> stack;
   stack.push(employees("1234","Yang",30000));
   stack.push(employees("5678","Rosen",28000));
   while(! stack.is_empty())
   {
      employees emp = stack.pop();
      emp.display();
   }
}
```

A binary search tree is composed of nodes, ordered by a key, with two links each, one to a left and one to a right subtree. The tree's advantage is quick access to a given node.

```
// treesort.cpp
// Reads all words in a text file, stores
// in binary tree, displays sorted list.
#include <iostream.h>
#include <string.h>
#include <fstream.h>
#include <stdlib.h>

class words
{
   char spelling[40];
   words *left,
         *right;
public:
   words()
      { *spelling = '\0'; left = right = NULL; }
   friend class word_trees;
};

class word_trees
{
   words *root;
public:
   word_trees() { root = NULL; }
   words *get_root() { return root; }
   void retrieve(char *filenm);
   void display(words *subtree);
   void insert(char *s,words *&nd);
   words *new_node(char *s);
};

void main()
{
   word_trees dictionary;
   cout << "\nText of file DECL_IND.DAT: \n";
   dictionary.retrieve("decl_ind.dat");
   cout << "\nText of same file, "
         << sorted alphabetically: \n";
   dictionary.display(dictionary.get_root());
}

void word_trees::display(words *subtree)
// Prints subtree thru inorder traversal.
{
   if (subtree != NULL)
   {
      if (subtree->left)
         display(subtree->left);
      cout << subtree->spelling << " ";
      if (subtree->right)
         display(subtree->right);
   }
}
// Class <word_trees>:

void word_trees::retrieve(char *filenm)
// Reads text file word by word, stores in tree using
<insert>.
{
   ifstream infile(filenm);
   if (infile.bad())
   {
      cout << filenm << " not found.\n";
      exit(1);  // in <stdlib.h>
   }
   char word[80];
   do
   {
      infile >> word;
      cout << word << " ";
      insert(word,root);
   }
   while (*word);
   infile.close();
   cout << endl;
}

void word_trees::insert(char *s,words *&nd)
// Creates node with string <s>,
// inserts it into tree below node <nd>.
{
   if (nd)
   {
      int strdiff = strcmp(s,nd->spelling);
      if (strdiff > 0)
         if (nd->right)
            insert(s,nd->right);
         else
            nd->right = new_node(s);
      else
         if (strdiff < 0)
            if (nd->left)
               insert(s,nd->left);
            else
               nd->left = new_node(s);
   }
   else
      nd = new_node(s);
}

words *word_trees::new_node(char *s)
// Allocates node, assigns <s> to it.
{
   words *node = new words;
   strcpy(node->spelling,s);
   return node;
}
```

Sample I/O of an elevator simulator

The listing below shows part of one sample run of the Deitel/Deitel solution to the elevator simulation problem. This listing may help you perform an object-oriented analysis of the problem by showing what some reasonable specifications for the problem are.

The only user input is the number of seconds of duration for the elevator simulation. Note that the arrival of persons is randomized in the application to simulate how an elevator is actually used.

```
Enter length of elevator simulation: 20
First person scheduled to arrive on floor 1 at time 15
First person scheduled to arrive on floor 2 at time 7
STARTING ELEVATOR SIMULATION
Elapsed time:    0
Elevator waiting on floor floor 1 for passengers
Elapsed time:    1
Elevator waiting on floor floor 1 for passengers
Elapsed time:    2
Elevator waiting on floor floor 1 for passengers
Elapsed time:    3
Elevator waiting on floor floor 1 for passengers
Elapsed time:    4
Elevator waiting on floor floor 1 for passengers
Elapsed time:    5
Elevator waiting on floor floor 1 for passengers
Elapsed time:    6
Elevator waiting on floor floor 1 for passengers
Elapsed time:    7
Person 1 created
Elevator called from floor 2
Person 1 walked onto floor 2 and pressed the down
  button
Next person scheduled to arrive on floor 2 at time 18
Bell reset
Light turned off
elevator door closed
floor door closed
Elevator starting to move up
Elapsed time:    8
Elevator moving up
Elapsed time:    9
Elevator moving up
Elapsed time:   10
Elevator moving up
Elapsed time:   11
Elevator moving up
Elapsed time:   12
Elevator stopped
floor door opened
Call button reset
elevator door opened
Bell sounded
Light turned on
Person 1 entered the elevator
Person leaving floor 2
Bell reset
Light turned off
elevator door closed
floor door closed
Elevator starting to move down
Elapsed time:   13
Elevator moving down
Elapsed time:   14
Elevator moving down
Elapsed time:   15
Elevator moving down
Person 2 created
Elevator called from floor 1
Person 2 walked onto floor 1 and pressed the up button
Next person scheduled to arrive on floor 1 at time 22
Elapsed time:   16
Elevator moving down
Elapsed time:   17
Elevator stopped
floor door opened
Call button reset
elevator door opened
Bell sounded
Person 1 exited the elevator
Light turned on
Person 2 entered the elevator
Person leaving floor 1
Bell reset
Light turned off
elevator door closed
floor door closed
Elevator starting to move up
Elapsed time:   18
Elevator moving up
Person 3 created
Elevator called from floor 2
Person 3 walked onto floor 2 and pressed the down
  button
Next person scheduled to arrive on floor 2 at time 31
Elapsed time:   19
Elevator moving up
Elapsed time:   20
Elevator moving up
```

A collection of points on a coordinate axis

A collection may be implemented in many ways: using an array of objects, a pointer to an array of objects, an array of pointers to objects, a linked list, a stack, a queue. The program below uses a class template to store a collection of *points* objects in a stack. The stack is implemented as a linked list.

```
// pointstk.cpp
// Implements collection of (x,y)-coordinate
// points as a stack, based on a linked list.
// Prompts for a series of points, displays
// in reverse order.
#include "liststac.h"

class points
{
public:
   points() { x = y = 0; }
   points(int x_init,int y_init)
      { x = x_init; y = y_init; }
   friend ostream& operator<<
      (ostream& os,points p);
private:
   int x,y;
};

ostream& operator<<(ostream& os,points p)
// Overloaded inserter.
{
   cout << "(" << p.x << ", " << p.y << ") ";
   return os;
}

void main()
{
   stacks<points> path;
   path.push(points(0,0));
   path.push(points(1,1));
   path.push(points(2,4));
   path.push(points(3,9));
   path.push(points(4,16));
   while (! path.is_empty())
      cout << path.pop() << endl;
}
```

Output:

```
(4, 16)
(3, 9)
(2, 4)
(1, 1)
(0, 0)
```

Stack class template library:

```
// liststac.h
// Defines stack class template
// using linked list
#include <iostream.h>

typedef int bool;

template <class T>
class list_nodes
{
public:
   list_nodes() { next = NULL; }
   list_nodes(T initval)
      { data = initval; next = NULL; }
   void display() { cout << data; }
   T get_data() { return data; }
   list_nodes* get_next() { return next; }
   void set_next(list_nodes* p)
      { next = p; }
private:
   T data;
   list_nodes* next;
};

template <class T>
class stacks
{
public:
   stacks() { first = NULL; }
   void push(T new_item)
   {
      list_nodes<T>* new_node =
         new list_nodes<T>(new_item);
      if (first != NULL)
         new_node->set_next(first);
      first = new_node;
   }
   T pop()  {
      T retval;
      if (! is_empty())
      {
         retval = first->get_data();
         first = first->get_next();
      }
      return retval;
      };
   bool is_empty()
      {  return (first == NULL); }
private:
   list_nodes<T>* first;
};
```

A class of strings of arbitrary length using inheritance

The class declared below uses inheritance to build on the functionality of the *strings* class, whose limit is 1000 characters. An *lstring* may have an unlimited size. It is implemented as a linked list.

Notice the header of the class declaration:

```
class lstrings : public strings
```

As a class that uses public inheritance, *lstrings* makes public members of its base class part of its public interface.

Conceptually, *lstrings* are a kind of *strings*. The designer of a class must decide whether inheritance is appropriate conceptually as well as technically. (Is a circle a kind of point? A point has a location and a circle has a location. But we don't think of a circle as a kind of point in day-to-day life. A long string, however, may be called a kind of string.)

```
// lstrings.h
// Safe string manipulation class
// library.

#include "strings.h"

class lstrings : public strings
{
   lstrings* next;
public:
   lstrings();
   lstrings(char* s);
   lstrings(strings s);
   boolean concat(char* s);
   boolean concat(lstrings s);
   int compare(char* s);
   unsigned int length();
   char nth_char(unsigned int n);
   char set_nth_char
      (unsigned int n,char ch);
   boolean insert
      (unsigned int loc, char* s);
   boolean delet
      (unsigned int loc, int num_chars);
   lstrings substring
      (unsigned int loc,int num_chars);
   boolean overwrite
      (unsigned int loc,char* s);
   void input();
   void display();
   friend ostream& operator<<
      (ostream& os,lstrings s);
};
```

The base class, *strings* (defined in an earlier example in the library file *strings.h*) has a single data member, a C-style string, *content*:

```
class strings
{
private:
    char content[STR_MAX];
public:
    strings();
    strings(char* s);
    ...
};
```

Most of the functions for *lstrings* are currently implemented in library file, *lstrings.cpp,* only as stubs. An exception is the conversion constructor that takes a C-style string as a parameter:

```
lstrings::lstrings(strings s)
// Conversion constructor from <char*>.
{
   assign(s.get_content());
}
```

This function must use a base-class access function, *get_content*, rather than directly passing the data member *content* as a parameter, because *content* is a private member of the base class *strings*. Derived classes do not have access to private members of their base classes. They do, however, have access to public and protected members. The difference between a protected member and a public one is that classes have access to protected members of their base classes, but to other code a protected member is inaccessible.

Challenge problem:

The class *strings* has a public access function, *get_content*, that returns a pointer to the *content* data member. Is it appropriate to make that public for *lstrings*, when it returns only *part* of the long string stored in an *lstrings* instance? Discuss two alternative implementations of the library that would change the access status of the base-class members.

General-purpose arrays using inheritance

Base-class declaration defines a generic array:

```
// array.h
// Generic array base class, available for inheritance.
#include <iostream.h>

class arrays
{
protected:
   int max_size,size;
   void **element;
public:
   arrays()
      { max_size = size = 0; }
   void set_max_size(int sz)
      { max_size = sz; }
   void *get_element(int subscript)
      { if (subscript < size) return element[subscript]; else return NULL; }
   void set_element(int subscript,void *value)
      { if (subscript < size) element[subscript] = value; }
   void insert(void *value)
      { if (size < max_size) element[size++] = value; }
};
```

max_size is capacity of array, *size* is current occupancy at any moment.

Double indirection is needed to dynamically allocate an array on the heap

Array's base type is *void* for generic collection; derived class will cast pointers to the collection's contained

---

The *arrays* base class may be used to create a derived class of string arrays:

```
// strarray.cpp
// Reads strings from keyboard, displays them.
// Uses generic array class <arrays>.
#include "array.h"
#include <string.h>

typedef char *char_ptrs;

class string_arrays: public arrays
{
public:
   string_arrays(int sz);
   char *str_ele(int subscript);
   void insert(char *s);
   void get();
   void display();
};

string_arrays::string_arrays(int sz)
// Constructor.
{
   arrays::arrays();
   set_max_size(sz);
   element = (void **)(new char_ptrs[sz]);
}

char *string_arrays::str_ele(int subscript)
// Returns array element <subscript>, a string.
{
   return (char *)get_element(subscript);
}
```

```
void string_arrays::insert(char *s)
// Adds contents of <s> to array.
{
   char *str = new char[strlen(s)+1];
   strcpy(str,s);
   arrays::insert(str);
}

void string_arrays::get()
// Prompts for a list of strings from user.
{
   int i = size;
   while (i < max_size) {
      char input[80];
      cout << "Enter a string (blank to quit): ";
      cin.getline(input,80);
      if (strlen(input) > 0)
         insert(input);
      else
         break;
      ++i;
   }
}

void string_arrays::display()
// Writes all strings to screen.
{
   for (int i=0; i < size; ++i)
      cout << str_ele(i) << endl;
}

void main()
{
   string_arrays A(10);
   A.get();
   A.display();
}
```

A virtual function's address is resolved at runtime

// animal.cpp

```
// Tells how animals move.
#include <iostream.h>

class animals
{
public:
   animals() {};
   void tell_motion() { cout << "\nAnimals move around.\n"; };
};

void main()
{
   animals a;
   a.tell_motion();
}
```

Output:

```
Animals move around
```

// bird1.cpp

```
// Tells how birds move.
#include <iostream.h>

class animals
{
public:
   animals() { };
   void tell_motion() { cout << "\nAnimals move around.\n"; };
};

class birds: animals
{
public:
   birds() { };
   void tell_motion()
      { cout << "\nBirds move on two legs and wings\n"; };
};

void main()
{
   birds b;
   b.tell_motion();
}
```

Output:

```
Birds move on two legs and wings
```

// bird2.cpp

```
// Tries to tell how birds move and reproduce.
#include <iostream.h>
class animals
{
public:
   animals() { };
   void tell_motion() { cout << "Animals move around.\n"; };
   void tell_reproduction() { cout << "Animals reproduce.\n"; };
   void tell_all() { tell_motion(); tell_reproduction(); };
};
class birds: public animals
{
public:
   birds() { };
   void tell_motion()
      { cout << "Birds move on two legs and fly.\n"; };
   void tell_reproduction()
      { cout << "Birds lay eggs.\n"; };
};
void main()
{
   cout << endl;
   birds b;
   b.tell_all();
}
```

Output:

```
Animals move around.
Animals reproduce.
```

// bird3.cpp

```
// Tells how birds move and reproduce. Uses virtual functions.
#include <iostream.h>
class animals
{
public:
   animals() { };
   void virtual tell_motion()
      { cout << "Animals move around.\n"; };
   void virtual tell_reproduction()
      { cout << "Animals reproduce.\n"; };
```

```
   void tell_all()
      { tell_motion(); tell_reproduction(); };
};

class birds: public animals
{
public:
   birds() { };
   void virtual tell_motion()
      { cout << "Birds move around on two legs and fly.\n"; };
   void virtual tell_reproduction()
      { cout << "Birds lay eggs.\n"; };
};

void main()
{
   cout << endl;
   birds b;
   b.tell_all();
}
```

Output:

```
Birds move around on two legs and fly.
Birds lay eggs.
```

Polymorphism in graphics

The program below illustrates how polymorphism and virtual functions permit us to create a collection of objects of mixed classes and run through this collection having each object call a function of the same name, *draw* here. Each object knows its own class, so it will call the virtual function *draw* of its own class to draw itself properly.

The program is based on the following inheritance hierarchy:

```
// shapes.cpp
// Defines an array of three drawable items
// (two lines, two dots), draws them on
// text screen.
// Declares <shapes> class and derived
// <dots> and <horiz_lines> classes.
#include <iostream.h>
#include <iomanip.h>

// The base class, an abstract class, has a pure
// virtual function <draw>, whose derived-class
// versions are called to draw each item.
class shapes
{
public:
   shapes() {};
   void set_x(int x) { x_origin = x; };
   int get_x() { return x_origin; }
   virtual void draw() = 0;
      // may not be called
private:
   int x_origin;
};

class dots : public shapes
{
public:
   dots(int x) { set_x(x); }
   virtual void draw()
      { cout << setw(get_x()) << "X" << endl; }
};

class horiz_lines : public shapes
{
public:
   horiz_lines(int x, int len)
      { set_x(x); length = len; }
   virtual void draw();
private:
   int length;
};

// Class <horiz_lines>:

void horiz_lines::draw()
// Puts rule of <length> at origin.
{
   cout << setw(get_x()) << "";
   for (int i=0; i < length; ++i)
         cout << "X";
   cout << endl;
}

//------------------

void main()
{
   // Spec an array of items:
   shapes *item[] =
   {
      new horiz_lines(3,10),
      new dots(4),
      new dots(5),
      new horiz_lines(6,20),
   };

   // Loop to draw each item:
   for (int i=0; i < (sizeof item) /
           (sizeof item[0]); ++i)
         item[i]->draw();
}
```

The output of this program is a line of 10 X's at horizontal position 3, a dot at position 4, a dot at position 5, and a line of 20 X's at position 6:

```
  XXXXXXXXXX
   X
    X
     XXXXXXXXXXXXXXXXXXXX
```

An application framework uses polymorphism

The application *wincalc.cpp* below uses an application framework defined in the library files *winfrmwk.h* and *winfrmwk.cpp*. The library defines a base class, *applications*, among others. The application *wincalc* uses an application class, *calculators,* derived from *applications*. The *calculators* class defines virtual functions *init_prompt, int_menu,* and *handle_event* that are all called from the base class's *run* function. The *run* function includes the event loop that controls the entire application.

```
// wincalc.cpp
// Repeatedly prompts for 2 integers,
// displays sum.
// Uses application framework defined in
// <winfrmwk.h>.
#include "winfrmwk.cpp"

class calculators : public applications
{
public:
   calculators() { };
   virtual void init_prompt()
      { set_prompt("Choose an operation"); };
   virtual void init_menu()
      { menu.set("+ Add","Q Quit",""); };
   virtual void handle_event(events event);
};

void main()
{
   // Declare an application, <calc>,
   // an instance of this program's class
   // <calculators> that inherits from
   // <applications> library class:
   calculators calc;
   // Run the application instance:
   calc.run();
}

void calculators::handle_event(events event)
// Virtual event handler for descendant of
// <applications>.
{
   // Handle base-class events, e.g.,
   // input of 'Q' to quit:
   applications::handle_event(event);

   // Handle events defined in this
   // derived class:
   switch(event.get_text())
   {
      case '+':
         cout << "Enter 2 integers: ";
         int input_1, input_2, sum;
         cin >> input_1 >> input_2;
         sum = input_1 + input_2;
         cout << input_1 << " + " << input_2
            << " = " << sum << endl;
         break;
   }
}
```

Sample I/O:

```
|------------------------------------|
| Choose an operation                |
|------------------------------------|
      + Add      Q Quit
+
Enter 2 integers: 2 5
2 + 5 = 7
|------------------------------------|
| Choose an operation                |
|------------------------------------|
      + Add      Q Quit
q
Goodbye
```

An application framework

```
// winfrmwk.h
// Declares text-mode library classes
// <events>, <windows>, <menus>,
// <applications>.
// David Keil, Framingham State College, 7/98
#ifndef WINFRMWK_H
#define WINFRMWK_H

#include <iostream.h>
#include <iomanip.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include <stdlib.h>

const int  MAX_WIN_HT = 20,  // in rows
           MAX_WIN_WIDTH = 80; // in columns

enum event_types
{ev_nothing,ev_escape,ev_menu_selection};

class events
{
public:
   events() { category = ev_nothing; };
   char get_text() { return text; };
   void fetch();
private:
   event_types category;
   char text;
};
class models
// Data model for storage in a window.
{
public:
   models() { num_lines = 0; }
   void set_line(char* s)
      { strcpy(line[0],s); num_lines++; }
   char* get_line(int line_num)
      { return line[line_num]; }
   void display(int line_num)
      { cout << line[line_num]; }
private:
   char line[MAX_WIN_HT-2][MAX_WIN_WIDTH-2];
   int num_lines;
};

// Class <windows>:

class windows
{
public:
   windows();
   windows(int x0,int y0,int w,int h,models* text);
   void set_line(char* s);
   int get_x() { return x; }
   int get_y() { return y; }
   int get_width() { return width; }
   int get_height() { return height; }
   void set(int x0,int y0,int w,int h);
   void draw_horiz();
   void draw_vert();
   void virtual draw();
   void move();
private:
   int x,y,// screen pos of upper-left corner
      width,height; // size of window
   models* content;
};

const int MAX_MENU_OPTIONS = 6,
          MENU_OPT_WID = 12; // Width of single label
in columns
```

```
class menus: windows
{
public:
   menus() { num_options = 0; };
   void set(char opt1[],...);
   void draw();
private:
   char option[MAX_MENU_OPTIONS][40];
   int num_options;
};

class applications
{
public:
   applications();
   void virtual init_prompt() { };
   void virtual init_menu() { menu.set(""); };
   void virtual handle_event(events event) {};
   void display_prompt();
   void set_prompt(char *s)
      { prompt_line.set_line(s); };
   void run();
private:
   windows prompt_line;
protected:
   menus menu;
};

#endif
```

Member function definitions for an application framework

```
// winfrmwk.cpp
// Linkable application-framework library.
// Defines member functions for windows,
// menus, application class.
// David Keil, Framingham State College, 7/98
#include "winfrmwk.h"

// Class <events>:

void events::fetch()
// Returns event record corresponding to keyboard
input.
{
   char input[80];
   cin >> input;
   switch (input[0])
   {
      case '\\':
         category = ev_escape;
         break;
      default:
         category = ev_menu_selection;
         text = input[0];
         break;
   }
}
// Class <windows>:

windows::windows()
{
   x = y = width = height = 0;
   content = NULL;
}

windows::windows(int x0,int y0,int w,
                 int h,models* text)
// Constructor.
{
   set(x0,y0,w,h);
   content = text;
}

void windows::set_line(char* s)
// Sets content.
{
   if (!content)
      content = new models;
   content->set_line(s);
}

void windows::set(int x0,int y0,int w,int h)
// Access function.
{
   x = x0; y = y0; width = w; height = h;
}

void windows::draw_horiz()
// Draws row of graphic characters across
// top.
{
   cout << "|";
   for (int col = 2; col < width-2; ++col)
      cout << '-';
   cout << "|" << endl;
}

void windows::draw_vert()
// Draws column of graphic characters
// on left.
{
   for (int row = 1; row < height-1; ++row)
   {
      if (content)
      {
         cout << "| ";
         content->display(row-1);
         int len =
            strlen(content->get_line(row-1));
         cout << setw(width - len - 3)
              << "|\n";
      }
   }
}

void windows::draw()
// Displays window.
{
   draw_horiz();
   draw_vert();
   draw_horiz();
}

// Class <menus>:

void menus::set(char opt1[], ...)
// Assigns parameters to option string array
// elements. Last arg must be null string.
{
   va_list ap;   // declared in <stdarg.h>
   va_start(ap,opt1);
   for (num_options=0; opt1[0] != '\0';
        ++num_options)
   {
      strcpy(option[num_options],opt1);
      opt1 = va_arg(ap,char *);
   }
   va_end(ap);
}

void menus::draw()
// Draws labels at top of screen.  4/96
{
   for (int i=0; i < num_options; ++i)
   {
      cout << setw(12) << option[i];
   }
   cout << endl;
}

// Class <applications>:

applications::applications()
// Default constructor.
{
   prompt_line.set_line("");
};

void applications::display_prompt()
// Shows prompt line in window.
{
   prompt_line.set(0,0,40,3);
   prompt_line.draw();
};
```

```
void applications::run()
// Executes event loop, terminates on "Q"
// for Quit.
{
   init_prompt();
   init_menu();
   events event;
   char event_text;
   do {
      display_prompt();
      menu.draw();
      event.fetch();
      event_text = toupper(event.get_text());
      if (event_text != 'Q')
      {
         handle_event(event);
      }
   }
   while (event_text != 'Q');
   cout << "\nGoodbye\n";
}
```

Examples related to exception handling

Standard C-style error handling

```
// div0.cpp
// Performs division on input values, trapping
// divide-by-zero errors locally.
#include <iostream.h>

float quotient(int a,int b);

void main()
{
   cout << "Enter two integers: ";
   int a,b;
   cin >> a >> b;
   cout << a << " / " << b << " = " << quotient(a,b) << endl;
}

float quotient(int a,int b)
// Returns (a/b). On divide-by-zero, shows message, returns 0.
{
   if (b != 0)
      return (float)a / b;
   else
   {
      cout << "Division by 0 is not permitted.\n";
      return 0;
   }
}
```

Error handling with *assert* terminates program prematurely

```
// div1.cpp
// Performs division on input values, using <assert> to
// trap divide-by-zero errors.
#include <iostream.h>
#include <assert.h>

float quotient(int a,int b);

void main()
{
   cout << "Enter two integers: ";
   int a,b;
   cin >> a >> b;
   cout << a << " / " << b << " = " << quotient(a,b) << endl;
}

float quotient(int a,int b)
// Returns (a/b). Terminates program on divide-by-zero.
{
   assert(b != 0);
   return (float)a / b;
}
```

C++-style exception handling separates error detection from a program's response

```
// div2.cpp
// Performs division on input values, using exception
// handling to manage divide-by-zero errors.
#include <iostream.h>
#include <string.h>

float quotient(int a,int b);

class math_errors
{
public:
   char message[80];
   math_errors(char *msg) { strcpy(message,msg); };
};

void main()
```

Programmer may define class such as this to use for throwing errors

These keywords are used in exception handling

```
{
   cout << "Enter two integers: ";
   int a,b;
   cin >> a >> b;
   try
   {
      cout << a << " / " << b << " = " << quotient(a,b) << endl;
   }
   catch (math_errors error)
   {
      cout << error.message << " is not permitted.\n";
   }
}

float quotient(int a,int b)
// Returns (a/b). Terminates program on divide-by-zero.
{
   if(b == 0)
      throw math_errors("Division by 0");
   return (float)a / b;
}
```

An iterator class for an integer collection

The program below declares a class, *int_collections,* to store an array of integers and its size. It is associated with a friend class, *int_iterators*. The iterator performs looping operations on a collection.

The *main* function of the program declares one instance of the collection and one of the iterator. The iterator retrieves a file of integers into the collection and displays the collection. It does so using two versions of a member function, *apply*, that accepts a pointer to a function and executes that function on each element of the collection.

```
// scorscan.cpp
// Defines array of scores, uses iterator class
// to retrieve and display it.
// David Keil, Framingham State College, 6/98
#include <iostream.h>
#include <fstream.h>

typedef int bool;
const bool false=0,true=1;

// Pointer-to-function data types:
typedef void (*functions_taking_int)(int);
typedef bool (*functions_taking_file)(int& n,ifstream&
infile);

// Utility functions:
bool retrieve(int& n,ifstream& infile);
void display(int n);

const int MAX_INTS = 10;

class int_collections
{
public:
   int_collections() { num_ints = 0; };
   friend class int_iterators;
private:
   int element[MAX_INTS];
   int num_ints;
};

class int_iterators
{
public:
   int_iterators(int_collections* p_arg);
   void first();
   bool next();
   void apply(functions_taking_int func);
   bool apply(functions_taking_file func,
      char* file_name);
private:
   int_collections* p_collection;
   int current;
};
void main()
{
   int_collections score_history;
   int_iterators scanner(&score_history);

   // Apply input and output routines to
   // each item in the collection:
   if (scanner.apply(retrieve,"scores.dat"))
   {
      cout << "Contents of SCORES.DAT:\n";
      scanner.apply(display);
      cout << endl;
   }
}

// Global functions:

bool retrieve(int& n,ifstream& infile)
// Reads from <infile> into <n>.
{
   if (infile.eof())
      return false;
   else
   {
      infile >> n;
      return true;
   }
}

void display(int n)
// Displays integer parameter.
{
   cout << n << " ";
}

// Class <int_iterators>:

int_iterators::int_iterators
   (int_collections* p_arg)
// Constructor.
{
   p_collection = p_arg;
   first();
}

void int_iterators::first()
// Sets index to point to first
// element in collection.
{
   current = 0;
}
bool int_iterators::next()
// Advances <current> if possible,
// reports success or failure.
{
   bool ok = (current < MAX_INTS-1);
   if (ok) ++current;
      return ok;
}

void int_iterators::apply
   (functions_taking_int func)
// Scans <db> array, applying <func>
// to each element.
{
   first();
   do
      func(p_collection->element[current]);
   while (next());
}

bool int_iterators::apply
   (functions_taking_file func,char* file_name)
// Opens file <file_name> and scans array,
// applying function <func> to each element
// of collection.
{
   ifstream infile(file_name,ios::nocreate);
   if (! infile)
   {
      cout << "File " << file_name
          << " not found" << endl;
      return false;
   }
   else
   {
      first();
      do {
         func(p_collection->element[current],
            infile);
      } while (next());
      return true;
   }
}

boolean int_iterators::next()
// Advances <current> if possible,
//reports success or failure.
{
   int ok = (current < MAX_INTS-1);
   if (ok)
      ++current;
   return ok;
}
```

```
void int_iterators::apply
   (functions_taking_int func)
// Scans <db> array, applying <func>
// to each element.
{
   first();
   do
      func(p_collection->element[current]);
   while (next());
}
```

Output:

```
Contents of SCORES.DAT:
1 2 3 4 5 6 7 8 9 10
```

Each iterator class is closely associated with a collection class. The iterator handles traversals of the collection or operations that must be performed on each item in the collection. An iterator has some features in common with a cursor or bookmark. The iterator class encapsulates a control aspect of a collection: looping. Separating the iterator class from the collection class has advantages; one collection may have two iterators scanning it at one time, for example.

The iterator is declared as a friend because it may be convenient to give the iterator access to private members.

The iterator object must know about both the collection it is associated with and about the *item* in the collection that is the iterator's current position in the collection; *p_collection* and *current* fill these requirements in the class *int_iterators*.

The code that uses the iterator must have a way to set the iterator to the first item in the collection; this is done by the function *first* in the example. The iterator steps to th enext item using the member function named *next*.

An *apply* function, a member of the iterator class, enables the programmer to scan the entire collection with one command, performing a particular operation on each item. The operation is specified by a pointer to the function that will carry it out, a global function. The iterator's member

```
void apply(functions_taking_int func);
```

for example, scans through the collection and causes the function *func* to be executed once for each item in the collection, with that item passed to *func* as a parameter. The programmer who uses the iterator determines what function will be passed.

The mechanism for passing functions a parameters is the pointer-to-function data type. One such type, *function_taking_int*, is declared in the example as follows:

```
typedef void(*function_taking_int)(int);
```

This *typedef* declares a type identifier, *function_taking_int,* and says that an instance of this type is a pointer to a void function that takes an *int* parameter.

A collection of dynamically allocated points

The program below shows how to implement a collection with pointers. It uses a class, *points*, to describe the objects that are to comprise the collection (points on a coordinate axis here). Another class, *point_collections,* encapsulates the collection concept, using an array of pointers and an integer. The integer records the current size of the collection.

As the program adds point items to the collection, it dynamically allocates point objects on the heap and assigns their addresses to the pointers in the array *point* of the class *point_collections*.

The program reads and displays the file *dynpoint.txt*, whose text is as follows:

3 9
0 0
2 4
1 1
4 16

The screen output is formatted by adding parentheses and commas.

The program is very similar to the code in *point.cpp*, presented as an example in topic 5 (Arrays). The class *points* is identical. The collection class *point_collections* is identical except it replaces the array of *points* with an array of pointers to *points*. The *retrieve* member function of *point_collections* uses dynamic allocation. The *display* member function dereferences pointers to display the objects in the collection.

Output:
(3,9) (0,0) (2,4) (1,1) (4,16)

```
// dynpoint.cpp
// Implements collection of (x,y)-coordinate
// points, using array of pointers to objects.
// Reads a series of points from file,
// displays them.
#include <bool.h>
#include <iostream.h>
#include <fstream.h>

class points
{
public:
   points() { x = y = 0; }
   points(int x_init,int y_init)
      { x = x_init; y = y_init; }
   bool input(istream& in)
      { return !!(in >> x >> y); }
   void display()
      { cout << "(" << x << "," << y << ") "; }
private:
   int x,y;
};

class point_collections
{
public:
   point_collections()
      { num_points = 0; }
   void retrieve(char* file_name);
   void display();
   enum { MAX_POINTS = 100 };
private:
   points* point[MAX_POINTS];
   int num_points;
};

void main()
{
   point_collections list;
   list.retrieve("dynpoint.txt");
   list.display();
}

// Class <point_collections>:

void point_collections::retrieve(char* file_name)
// Reads from file <file_name> into collection.
{
   ifstream fin(file_name,ios::nocreate);
   points buffer;
   while(buffer.input(fin))
   {
      point[num_points] = new points;
      *(point[num_points]) = buffer;
      num_points++;
   }
   fin.close();
}

void point_collections::display()
// Outputs all items to screen.
{
   for (int i=0; i < num_points; ++i)
      point[i]->display();
   cout << endl;
}
```

A collection of integers using a dynamically allocated array

```
/*
    grocery.c
    Prompts for and displays a list of
    amounts for grocery bills.
    Uses abstract data type with dynamically
    allocated   integer array to store list.
*/
#include <stdio.h>
#include <stdlib.h>

/* Define abstract data type Groc_list: */

struct Groc_list_tag
{
    int* bill;
    int count;
};
typedef struct Groc_list_tag Groc_list;

void GL_init(Groc_list* plist);
void GL_input(Groc_list* plist);
void GL_add_item(Groc_list* plist,int item);
void GL_display(Groc_list list);

void main()
{
    Groc_list list;

    GL_init(&list);
    GL_input(&list);
    GL_display(list);
}

/* Operations associated with ADT
    Groc_list: */

void GL_init(Groc_list* plist)
/* Constructor; initializes members. */
{
    plist->bill = NULL;
    plist->count = 0;
}

void GL_input(Groc_list* plist)
/* Prompts user for member values. */
{
    int input,done=0;
    while (! done)
    {
       printf("Enter bill amount: ");
       scanf("%i",&input);
       if (input > 0)
          GL_add_item(plist,input);
       else
          done = 1;
    }
}

void GL_add_item(Groc_list* plist,int item)
/* Inserts a value, <item>, into list,
    allocating an array on the heap of
    sufficient size for all items and
    deallocating previous array. */
{
    int i, *new_array, bytes;

    /* Allocate space on heap: */
    bytes = sizeof(int) * (plist->count + 1);
    new_array = (int*)malloc(bytes);
    if (! new_array)
    {
       printf("Out of memory.\n");
       exit(0);
    }

    /* Insert new value at end of array: */
    new_array[plist->count] = item;

    /* Copy old array to newly allocated
        array: */
    if (plist->count > 0)
    {
       for (i = 0; i < plist->count; ++i)
          new_array[i] = plist->bill[i];
       free(plist->bill);
    }

    /* Update members of the structure
     pointed to by the parameter <plist>: */
    plist->bill = new_array;
    ++plist->count;
}

void GL_display(Groc_list list)
/* Show values of grocery bills stored
    in array: */
{
    int i;
    for (i=0; i < list.count; ++i)
       printf("%i ",list.bill[i]);
}
```

Suggested exercises:

1. Modify the program to define and call a function that takes a grocery list parameter and calculates the total of all grocery bills and the average bill.
2. Memory leaks occur when dynamically allocated variables become inaccessible. The pointer *new_array* in *GL_add_item* points to a dynamically allocated array on the heap. The variable *new_array*, being local, disappears when its function terminates. Why is this not a memory leak?
3. Rewrite the program in C++, making use of (a) member functions and (b) operators associated with dynamic allocation.

A linked list implemented using an array

An array implementation of linked lists stores the link member of a node as a subscript, rather than a pointer. The diagram below shows the way subscripts in link nodes tell where the next array element (node) in the linked list is. The list adds to itself by inserting new nodes in alphabetical order.

```
// arrlist.cpp
// Accepts four names, stores them in an array whose
// elements are nodes of a linked list, links nodes
// alphabetically using subscripts, displays list.
// Each link member of a node is the subscript of
// the next node.  If <next> value is (-1), node is
// last in list.
#include <iostream.h>
#include <string.h>

const int MAX_RECS = 4,
          MAX_NAME_SIZE = 20;

struct nodes
{
   char name[MAX_NAME_SIZE];
   int next;
};

class arr_lists
{
   nodes node[MAX_RECS];
   nodes header;
   int num_recs;
public:
   arr_lists();
   void read();
   void show();
private:
   int predecessor(char *value);
   void insert_node(char *value);
};

void main()
{
   arr_lists name_list;
   name_list.read();
   name_list.show();
}

arr_lists::arr_lists()
// Default constructor.
{
   num_recs = 0;
   for (int i=0; i < MAX_RECS; ++i)
   {
      node[i].name[0] = '\0';
      node[i].next = -1;
                        // Sentinel value for end-of-list link.
   }
   header.next = -1;
}
void arr_lists::read()
// Accepts a series of names from keyboard, inserts
// into array/linked list in alphabetical order.
{
   while(num_recs < MAX_RECS)
   {
      cout << "Next name: ";
      char input[MAX_NAME_SIZE];
      cin >> input;
      insert_node(input);
   }
}

void arr_lists::show()
// Displays list in alphabetical order.
{
   int i = header.next;
   while (i != -1)
   {
      cout << node[i].name << endl;
      i = node[i].next;
   }
}

int arr_lists::predecessor(char *value)
// Traverses list, returns subscript of <value>'s
// predecessor, alphabetically, or (-1) if <value>
// is lower than any list element.
{
   int pred = -1,i = header.next;
   while (i != -1)
   {
      // If <value> should be before <i>th node,
      // return (pred):
      if (strcmp(value,node[i].name) < 0)
         return pred;
      pred = i;
      i = node[i].next;
   }
   return pred;
}

void arr_lists::insert_node(char *value)
// Adds element with string <value> to array, links it
// to next-lower and next-higher value alphabetical
// order.
{
   // Insert:
   strcpy(node[num_recs].name,value);
   // Link in alphabetical order:
   int pred = predecessor(value);
   if (pred == -1)  // Insert as first node
   {
      int second = header.next;
      header.next = num_recs;
      node[num_recs].next = second;
   }
   else // Insert after predecessor:
   {
      int successor = node[pred].next;
      node[pred].next = num_recs;
      node[num_recs].next = successor;
   }
   ++num_recs;
}
```

Output:

```
Next name: tom
Next name: jill
Next name: carol
Next name: ann
ann
carol
jill
tom
```

A linked list of strings

The program below defines a singly linked list class, *linked_lists*. Each node of the list is of the type *list_nodes* and contains a string and a pointer to the next node. The linked-list object will contain one node whose *next* pointer points to the first node in the list that contains a data value.

```
// strlist.cpp
// Builds a linked list of strings from user
// input, displays it, prompts for deletions,
// displays result.
#include <iostream.h>
#include <string.h>
#include <ctype.h>

class list_nodes
{
public:
    list_nodes() { next = NULL; };
    list_nodes(char *s)
        { strcpy(value,s); next = NULL; };
    char *get_value() { return value; };
    list_nodes *get_next() { return next; };
    friend class linked_lists;
private:
    char value[80];
    list_nodes *next;
};

class linked_lists
{
public:
    linked_lists() { header.next = NULL; };
    void prepend(char *s);
    void delete_node(list_nodes *pred);
    void display();
    list_nodes *first() { return header.next; };
    list_nodes *get_header()
        { return &header; };
private:
    list_nodes header;
};

void linked_lists::prepend(char *s)
// Inserts a node containing data value <s>
// at the beginning of a linked list.
{
    list_nodes *new_node = new list_nodes(s);
    if (header.next == NULL)
        header.next = new_node;
    else
    {
        new_node->next = header.next;
        header.next = new_node;
    }
}

void linked_lists::delete_node (list_nodes *pred)
// Deletes the node following <pred> from list.
{
    // Take no action on null parameter:
    if (pred == NULL)
        return;

    // Record address of node to be deleted:
    list_nodes *deleted_node = pred->next;

    // Link predecessor of deleted node
    // to successor of deleted node:
    pred->next = deleted_node->next;
}

void linked_lists::display()
// Outputs data values of all nodes of list.
{
    // Counter is stored in pointer:
    list_nodes *p_node = header.next;

    // Loop through list, displaying
    // data member:
    while (p_node != NULL)
    {
        cout << p_node->value << endl;
        p_node = p_node->next;
    }
}

void main()
{
    // Build list from user input, and display:
    linked_lists list;
    char input[80];
    cout << endl << endl;
    do {
        cout << "Enter a string (* to quit): ";
        cin >> input;
        if (strcmp(input,"*") != 0)
            list.prepend(input);
    } while (strcmp(input,"*") != 0);
    list.display();

    // Prompt for deletions and display result:
    list_nodes *p_node = list.first(),
               *p_prev = list.get_header();
    while (p_node != NULL)
    {
        cout << "Delete " << p_node->get_value()
             << "?\n";
        char input;
        cin >> input;
        if (toupper(input == 'Y'))
            list.delete_node(p_prev);
        else
            p_prev = p_node;
        p_node = p_node->get_next();
    }
    cout << "\nWhat remains is:\n";
    list.display();
}
```

A stack of integers (topic 9)

The program below implements the stack data structure as an array.

```
// intstack.cpp
// Pushes a series of integers onto a stack,
// pops them to display in reverse order.
// David Keil, Framingham State College, 9/98
#include <iostream.h>
#include <assert.h>

typedef int bool;

const int MAX_STACK_SZ = 20;

class stacks
{
public:
    stacks() { size = 0; };
    void push(int n)
        {
            if (! is_full())
                element[size++] = n;
        };
    int pop()
        {
            assert(! is_empty());
            return element[--size];
        };
    bool is_empty()
        {
            return (size == 0);
        };
    bool is_full()
        {
            return (size >= MAX_STACK_SZ);
        };
private:
    int element[MAX_STACK_SZ];
    int size;
};

void main()
{
    stacks stack;
    stack.push(1);
    stack.push(2);
    stack.push(3);
    stack.push(4);
    while (! stack.is_empty())
        cout << stack.pop() << endl;
}
```

Output:
4 3 2 1

A stack of points (topic 9)

The program below implements the stack data structure using a linked list. It reads from the following data file, *pointstk.txt*:

```
3 9
0 0
2 4
1 1
4 16
```

```
// pointstk.cpp
// Implements collection of (x,y)-coordinate
// points, using stack based on linked list.
// Reads a series of points from file,
// stores in stack, displays in reverse order.
// David Keil, Framingham State College, 11/98
#include <iostream.h>
#include <fstream.h>
#include <assert.h>

class points
{
public:
    points() { x = y = 0; }
    points(int x_init,int y_init)
        { x = x_init; y = y_init; }
    bool input(istream& in)
        { return !!(in >> x >> y); }
    void display()
        { cout << "(" << x << "," << y << ") "; }
private:
    int x,y;
};

class point_nodes // list node type
{
public:
    point_nodes()
        { next = NULL; }
    point_nodes(points pt,point_nodes* nx)
        { point = pt; next = nx; }
    point_nodes* get_next()
        { return next; }
    points get_point()
        { return point; }
    void set_next(point_nodes* nx)
        { next = nx; }
private:
    points point;
    point_nodes* next;
};

class point_stacks
{
public:
    point_stacks()
        { num_points = 0; }
    void push(points pt);
    points pop();
    bool is_empty()
        { return (header.get_next() == NULL); }
private:
    point_nodes header;
    int num_points;
};

void main()
{
    point_stacks stack;
    ifstream fin("pointstk.txt",ios::nocreate);
    points point;
    while(point.input(fin))
        stack.push(point);
    fin.close();
    while (! stack.is_empty())
        point = stack.pop();
}

// Class <point_stacks>:

void point_stacks::push(points pt)
// Inserts <pt> on top of stack.
{
    cout << "Pushing ";
    pt.display();
    cout << endl;
    point_nodes* new_node =
        new point_nodes(pt,header.get_next());
    header.set_next(new_node);
}

points point_stacks::pop()
// Removes top point from stack, returns it.
// On empty stack, terminates program.
{
    assert(! is_empty());
    point_nodes* p_top = header.get_next();
    points pt = p_top->get_point();
    header.set_next(p_top->get_next());
    cout << "Popping ";
    pt.display();
    cout << endl;
    return pt;
}
```

Output:

```
Pushing (3,9)
Pushing (0,0)
Pushing (2,4)
Pushing (1,1)
Pushing (4,16)
Popping (4,16)
Popping (1,1)
Popping (2,4)
Popping (0,0)
Popping (3,9)
```