Transparency Masters

Section 1.1: A First C Program

```
/*   This program prints the message
         ANSI--A kinder and gentler C!
     to the video display.
*/

#include <stdio.h>

main()
{
   printf( "ANSI--A kinder and gentler C!\n" );
}
```

Section 1.2: Conversion of Lengths

```
/*   This  program reads a length in yards from the keyboard,
     converts the  length  to  feet and inches, and writes
     the converted length to  the  video  display.  It  is
     assumed  that  the  length is a nonnegative  integer.
     The  program  terminates  when  a negative integer is
     entered. */

#include <stdio.h>

main()
{
     int yard, foot, inch;

     printf( "Enter next length " );
     scanf( "%d", &yard );

     while ( yard >= 0 ) {
          foot = 3 * yard;
          inch = 36 * yard;
          printf( "\n%d yd =\n", yard );
          printf( "%d ft\n", foot );
          printf( "%d in\n", inch );
          printf( "Enter next length " );
          scanf( "%d", &yard );
     }

     printf( "*** END OF PROGRAM ***\n" );
}
```

Section 1.10: Computing Income Tax

```
/*   This  program  reads  incomes from the file incomes.dat
     until end-of-file and writes the taxes due to the file
     out.dat. Income under  6000 greenbacks is taxed at 30
     percent, and income greater than  or  equal  to  6000
     greenbacks  is taxed at 60 percent. We assume  that
     the  income is an integer. The tax is written as an
     integer. */

#include <stdio.h>

main()
{
     int income, tax;
     FILE *fin, *fout;

     fin = fopen( "incomes.dat", "r" );
     fout = fopen( "out.dat", "w" );

     while ( fscanf( fin, "%d", &income ) != EOF ) {
          fprintf( fout, "Income = %d greenbacks\n", income );
          if ( income < 6000 )
               tax = 0.3 * income;
          else
               tax = 0.6 * income;
          fprintf( fout, "Tax = %d greenbacks\n", tax );
     }
}
```

Section 2.5: Statistical Measures

```
/* This program reads a series of real numbers from the
   standard input and prints a statistical summary of the
   data that includes the largest value input, the smallest
   value input, the sum of all values read, the mean
   (average) of all values input, the population variance,
   and the standard deviation. */

#include <stdio.h>
#include <math.h>

main()
{
     float x, /* current value input */
           max, /* largest value seen so far */
           min, /* smallest value seen so far */
           sum, /* sum of all values seen so far */
           mean, /* mean of all values read */
           sum_of_squares, /* sum of squares of all
                              values seen so far */
           variance; /* variance of all values read */
     int   count; /* number of values read so far */
     if ( scanf( "%f", &x ) == EOF )  /* empty file? */
          printf( "0 data items read\n" );
     else {
          max = min = sum = x;
          count = 1;
          sum_of_squares = x * x;
          while ( scanf( "%f", &x ) != EOF ) {
               count += 1;
               if ( x > max )  /* new high? */
                    max = x;
               if ( x < min )  /* new low? */
                    min = x;
```

```
            sum += x;
            sum_of_squares += x * x;
        }
        printf( "%d data items read\n", count );
        printf( "maximum value read = %f\n", max );
        printf( "minimum value read = %f\n", min );
        printf( "sum of all values read = %f\n", sum );
        mean = sum / count;
        printf( "mean = %f\n", mean );
        variance = sum_of_squares / count - mean * mean;
        printf( "variance = %f\n", variance );
        printf( "standard deviation = %f\n",
            sqrt( variance ) );
    }
}
```

Section 2.8: Printing a Bar Graph

```
/*   This program prints the names of comic books and a bar
     graph to show the values of the books in thousands of
     dollars. */

#include <stdio.h>

main()
{
     char c;
     int value; /* value of comic */
     int i;  /* loop counter */

     while ( scanf( " %c", &c ) != EOF ) {
          do { /* print title of comic */
               printf( "%c", c );
               scanf( "%c", &c );
          } while ( c != '\n' );
          printf( "\n" );
          scanf( "%d", &value );  /* get value of comic */
          /* print value / 2 stars */
          for ( i = 1; i <= value / 2; i++ )
               printf( "*" );
          printf( "\n" );
     }
}
```

Section 3.2: Generating Prime Numbers

```
/* This program prints all positive prime integers less than
   or equal to the user-supplied value n. */

#include <stdio.h>

main()
{
     int possible_prime, n, possible_divisor;

     printf( "\tThis program lists all primes <= n\n\n" );
     printf( "Input n: ");
     scanf( "%d", &n );
     printf( "\n\n\tPrimes <= %d:\n\n", n );

     for ( possible_prime = 2; possible_prime <= n;
               possible_prime++ ) {
          /* try to find a divisor of possible_prime */
          for ( possible_divisor = 2;
               possible_divisor < possible_prime;
               possible_divisor++ )
               if ( possible_prime % possible_divisor == 0 )
                    /* found a divisor so possible_prime is
                       not prime */
                    break;
          if ( possible_divisor == possible_prime )
               /* exhausted possible divisors, so
                  possible_prime is prime */
               printf( "%d\n", possible_prime );
     }
}
```

Section 3.6: Printing a Calendar

```
/*   This program prints a calendar for a year. The user
     enters a code for the day of week for Jan 1 to begin
     and a code to indicate whether or not the calendar is
     to be a leap year calendar.

     The codes are: day_code (0 = Sun, 1 = Mon, etc.)
                    leap_year (0 = no leap year,
                               1 = leap year)   */

#include <stdio.h>

main()
{
     int day_code, /* day_code = 0: the month starts on Sun
                      day_code = 1: the month starts on Mon
                      day_code = 2: the month starts on
                                    Tues, etc.  */
         days_in_month,     /* number of days in month
                               currently being printed */
         leap_year, /* 1 means leap year; 0 means no leap
                       year */
         day,       /* counter for day of month */
         month;     /* month = 1: Jan, month = 2: Feb, etc. */

     do {
          printf( "Enter day and leap year codes: " );
          scanf( "%d%d", &day_code, &leap_year );
     } while ( day_code < 0 || day_code > 6 );

     for ( month = 1; month <= 12; month++ ) {
          switch ( month ) { /* print name and set
                                days_in_month */
          case 1:
```

```
          printf( "\n\nJanuary" );
          days_in_month = 31;
          break;
     case 2:
          printf( "\n\nFebruary" );
          days_in_month = leap_year ? 29 : 28;
          break;
     case 3:
          printf( "\n\nMarch" );
          days_in_month = 31;
          break;
     case 4:
          printf( "\n\nApril" );
          days_in_month = 30;
          break;
     case 5:
          printf( "\n\nMay" );
          days_in_month = 31;
          break;
     case 6:
          printf( "\n\nJune" );
          days_in_month = 30;
          break;
     case 7:
          printf( "\n\nJuly" );
          days_in_month = 31;
          break;
     case 8:
          printf( "\n\nAugust" );
          days_in_month = 31;
          break;
     case 9:
          printf( "\n\nSeptember" );
          days_in_month = 30;
          break;
```

```
        case 10:
             printf( "\n\nOctober" );
             days_in_month = 31;
             break;
        case 11:
             printf( "\n\nNovember" );
             days_in_month = 30;
             break;
        case 12:
             printf( "\n\nDecember" );
             days_in_month = 31;
             break;
        }

        printf( "\n\nSun  Mon  Tue  Wed  Thu  Fri  Sat\n" );

        /* advance printer to correct position for first
           date */
        for ( day = 1; day <= 1 + day_code * 5; day++ )
             printf( " " );

        /* print the dates for one month */
        for ( day = 1; day <= days_in_month; day++ ) {
             printf( "%2d", day );
             /* before Sat? */
             if ( ( day + day_code ) % 7 > 0 )
                  /* move to next day in same week */
                  printf( "   " );
             else  /* skip to next line to start with Sun */
                  printf( "\n " );
        }
        /* set day_code for next month to begin */
        day_code = ( day_code + days_in_month ) % 7;
    }
}
```

Example 4.1.1: Assigning a Grade

```
#include <stdio.h>
char grade( int exam1, int exam2, float exam_weight );
main()
{
     int ans, mid_term, final;
     float weight;
     char letter_grade;

     do {
          printf( "\n\nCompute another grade "
                  "(1 = Yes, 0 = No)? " );
          scanf( "%d", &ans );
          if ( ans == 1 ) {
               printf( "\nEnter mid term, final, weight: " );
               scanf( "%d%d%f", &mid_term, &final, &weight );
               letter_grade = grade( mid_term, final,
                                     weight );
               printf( "Grade is: %c\n", letter_grade );
          }
     } while ( ans == 1 );
}

char grade( int exam1, int exam2, float exam1_weight )
{
     float average;

     average = exam1_weight * exam1 +
               ( 1.0 - exam1_weight ) * exam2;
     if ( average > 7.0 )
          return 'P';
     else
          return 'F';
}
```

Section 4.4: Computing Resistance

```
/*   This program computes the resistance in ohms for a
     resistor. It prints a menu of colors on bands that mark
     a resistor and asks the user to enter character codes
     for each brand; for example, the user might enter 'B'
     for black. The program decodes the characters, computes
     the resistance, and prints the result.

     The function main invokes print_codes to print the codes,
     getchar to read three such codes from the keyboard, and
     decode_char to do the decoding. main uses a standard
     library function pow, which raises a number to a power
     to help compute the resistance; main then prints the
     resistance as a floating-point number. */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void    print_codes( void );     /* menu of codes */
double  decode_char( char code );

main()
{
     char    code1, code2, code3;     /* one code per band */
     double  resistance;
     double  color1, color2, color3;  /* decoded values */
     int     flag;

     /* Print codes and prompt for user input. */
     print_codes();
     printf( "\n\n\tEnter three codes. " );

     /* Read three character codes. */
```

```
     code1 = getchar();
     code2 = getchar();
     code3 = getchar();

     /* Decode each character code. */
     color1 = decode_char( code1 );
     color2 = decode_char( code2 );
     color3 = decode_char( code3 );

     /* Check whether codes were legal. */
     if ( color1 == -999.0  ||  color2 == -999.0  ||
          color3 == -999.0 )
          printf( "\n\n\tBad code -- cannot compute "
                  "resistance\n" );
     /* If codes were legal, compute and print resistance in
        ohms. */
     else {
          resistance = ( 10.0 * color1  +  color2 )
                       * pow( 10.0, color3 );
          printf( "\n\n\tResistance in ohms:\t%f\n",
                  resistance );
     }

     return EXIT_SUCCESS;
}

/*   This function prints a menu of color codes to guide the
     user in entering input. */

void  print_codes( void )
{
   printf( "\n\n\tThe colored bands are coded as "
           "follows:\n\n\n\t" );
   printf( "COLOR\t\t\tCODE\n\t" );
   printf( "-----\t\t\t----\n\n" );
```

```
   printf( "\tBlack-------------------> B\n" );
   printf( "\tBrown-------------------> N\n" );
   printf( "\tRed---------------------> R\n" );
   printf( "\tOrange------------------> O\n" );
   printf( "\tYellow------------------> Y\n" );
   printf( "\tGreen-------------------> G\n" );
   printf( "\tBlue--------------------> E\n" );
   printf( "\tViolet------------------> V\n" );
   printf( "\tGray--------------------> A\n" );
   printf( "\tWhite-------------------> W\n" );
}

/*   This function expects a character (color code) and
     returns a double precision floating point number as its
     value. If the code is not legal, it returns a value that
     signals this fact. */

double decode_char( char code )
{
     switch ( code ) {
     case 'B':
          return 0.0;
     case 'N':
          return 1.0;
     case 'R':
          return 2.0;
     case 'O':
          return 3.0;
     case 'Y':
          return 4.0;
     case 'G':
          return 5.0;
     case 'E':
          return 6.0;
     case 'V':
```

```
        return 7.0;
    case 'A':
        return 8.0;
    case 'W':
        return 9.0;
    default:
        return -999.0;     /* illegal code */
    }
}
```

Section 4.7: Simulating a Dice Game

```
/* main.h -- #define's included in main */

#define   TRUE      1
#define   FALSE     0
#define   NL        putchar( '\n' )
#define   TAB       putchar( '\t' )
#define   FORMAT    NL, NL, TAB
#define   FORMAT2   NL, TAB, TAB
#define   NESTEGG   500
#define   LIMIT     250

#define   PLAY_PROMPT      FORMAT, \
            printf( "Want to play? (1: yes, 0: no)  " )
#define   READ_ANS( ans )  scanf( "%d", (&ans) )

#define   BET_PROMPT       FORMAT, \
            printf( "Your bet in dollars?  " )
#define   BET_REPROMPT     FORMAT2, \
            printf( "Please enter your bet again.  ")
#define   READ_BET( bet )  scanf( "%d", (&bet) )

#define   WARN1( bet,lmt ) FORMAT, TAB, \
            printf( "\aYour bet of %d exceeds the limit " \
                    "of %d.", \
                    (bet), (lmt) )
#define   WARN2( bet,bnk ) FORMAT, TAB, \
            printf( "\aYou bet %d; you have %d.", (bet), \
                    (bnk) )

#define   WINNER_MSG       FORMAT, TAB, \
            printf( "You're winning -- don't quit now!" )
#define   LOSER_MSG        FORMAT, TAB, \
            printf( "You're losing -- don't quit now!" )
```

```
#define   BANKROLL_MSG( r ) FORMAT2, \
            printf( "Your bankroll stands at %d", (r) )

/* end main.h */

/* roll.h -- #defines included in roll_them_bones */

#define  SNAKE_EYES        2
#define  BOX_CARS         12

#define   PRINT_MARK( mark )     NL, FORMAT2, \
            printf( "Your current mark is: %d.", (mark) )
#define   PRINT_ROLL_MSG         FORMAT2, \
            printf( "Roll 'em again!" )
#define   PRINT_DICE( dice )     FORMAT2, \
            printf( "You rolled a %d.", (dice) )
#define   LOSE_MSG( bet )        FORMAT2, \
            printf( "Take %d dollars, but don't run!", \
                          (bet) )
#define   POSSIBLE_DOUBLE_MSG    FORMAT2, \
            printf( "Win and you win double!" )
#define   DOUBLE_WIN_MSG( amt )  FORMAT2, \
            printf( "You win double -- %d dollars!", (amt) )
#define   REGULAR_WIN_MSG( amt ) FORMAT2, \
            printf( "You win %d dollars.", (amt) )
#define   CONSOLE_MSG( bet )     FORMAT2, \
            printf( "You lost a trifling %d bucks.", (bet) )
#define   SHUT_EYES 3

/* end roll.h */

/*             main program              */

#include <stdio.h>
#include <stdlib.h>
```

```
#include <limits.h>
#include "main.h"

/* roll_them_bones simulates a game & returns a new
   bankroll  */
int roll_them_bones( int bet, int bankroll );

/* pause for time proportional to how_long */
void sleep( int how_long );

/* random number generator: ran(k) returns
   an integer in the range 1 through k */
int ran( int k );

main()
{
     int    bankroll = NESTEGG;
     int    ans;           /* 1: play; 0: halt */
     int    bet;           /* player's bet in dollars */

     /* Play until desire or money runs out. */
     do {
          BANKROLL_MSG( bankroll );
          PLAY_PROMPT;
          READ_ANS( ans );
          if ( ans == TRUE ) {
               BET_PROMPT;
               READ_BET( bet );
               while ( bet > LIMIT ) {
                    WARN1( bet, LIMIT );
                    BET_REPROMPT;
                    READ_BET( bet );
               }
               while ( bet > bankroll ) {
                    WARN2( bet, bankroll );
```

```
                    BET_REPROMPT;
                    READ_BET( bet );
               }

               /* Update bankroll after a game's played. */
               bankroll = roll_them_bones( bet, bankroll );
          }
     } while ( ans == TRUE && bankroll > 0 );

     if ( bankroll > 0 ){
          WINNER_MSG;
          BANKROLL_MSG( bankroll );
     }
     else
          LOSER_MSG;

     return EXIT_SUCCESS;
}

/*  ROLL_THEM_BONES -- simulates one game   */

#include "roll.h"

int roll_them_bones( int bet, int bankroll )
/* bet in dollars */
/* bankroll in dollars */
{
     int  mark;   /* sum of dice on 1st roll */
     int  dice;   /* sum of dice after 1st roll */

     /* TRUE if dice equals 2 or 12 */
     int  double_flag = FALSE;

     /* Roll the dice the first time to get the player's
        "mark". */
```

```
mark = ran( 6 ) + ran( 6 );
PRINT_MARK( mark );

/* Check for 7 or 11. */
if ( mark == 7  ||  mark == 11 ) {
     LOSE_MSG( bet );
     bankroll += bet;
     return bankroll;   /* Exit a winner. */
}

/* Check for possibility of double payoff on bet. */
if ( mark == SNAKE_EYES  ||  mark == BOX_CARS ) {
     double_flag = TRUE;
     POSSIBLE_DOUBLE_MSG;
}

/* Roll until DICE equals either MARK or 7 or 11. */
do {
     dice = ran( 6 ) + ran( 6 );
     PRINT_ROLL_MSG;
     NL;
     sleep( SHUT_EYES ); /* pause a bit to simulate pause
                            between rolls of the dice. */
     PRINT_DICE( dice );
} while ( dice != mark  &&  dice != 7  &&  dice != 11 );

if ( dice == mark )
     if ( double_flag ) {
          bankroll += 2 * bet;
          DOUBLE_WIN_MSG( 2 * bet );
     }
     else {
          bankroll += bet;
          REGULAR_WIN_MSG( bet );
     }
```

```
     else {
          bankroll -= bet;
          CONSOLE_MSG( bet );
     }
     return bankroll;
}

/*   ran returns a random integer in the range 1 to k.

     ran invokes the library function rand which returns a
     random integer in the range 0 to RAND_MAX (defined in
     stdlib.h). */

int ran( int k )
{
     double x = RAND_MAX + 1.0;
     int y;

     y = 1 + rand() * (k / x);
     return y;
}

/* sleep pauses for time proportional to how_long */
void sleep( int how_long )
{
     int i, j;

     for ( i = 0; i < how_long; ++i )
          /* INT_MAX is defined in limits.h */
          for ( j = 0; j < INT_MAX; ++j )
               ;
}
```

Example 4.8.1: Factorial Function

```
#include <stdio.h>
#include <stdlib.h>

int fact( int num );

main()
{
     int num;

     printf( "\nPlease enter a number:  " );
     scanf( "%d", &num );

     if ( num < 0 )      /* legal input? */
        printf( "\nERROR -- number must be >= 0." );
     else
        printf( "\nFactorial of %d is %d.", num,
                fact( num ) );

     return EXIT_SUCCESS;
}

int fact( int num )
{
     if ( num <= 1 )          /* base cases */
        return 1;
     else                     /* recursive call */
        return num * fact( num - 1 );
}
```

Example 4.8.2: Tower of Hanoi

```
#include <stdio.h>
#include <stdlib.h>

void  hanoi( char peg1, char peg2, char peg3, int how_many );

main()
{
     char  peg1 = 'A',   /* origin */
           peg2 = 'B',   /* destination */
           peg3 = 'C';   /* spare */

     /* number of disks initially on the origin */
     int   how_many;

     /* Prompt for and warn about the number of disks to be
        moved. */
     printf( "\n\tHow many disks initially on peg A?"
             "\n\tIf more than 7, the solution may seem to "
             "take forever!\n\t" );
     scanf( "%d", &how_many );

     /* Anything to move? */
     if ( how_many < 1 )
          printf( "\n\tThere's nothing to move!" );
     /* Otherwise solve with:                        */
     /*   -- peg1 as the origin                      */
     /*   -- peg2 as the destination                 */
     /*   -- peg3 as the spare                       */
     else
          hanoi( peg1, peg2, peg3, how_many );

     return EXIT_SUCCESS;
}
```

```
void  hanoi( char p1, char p2, char p3, int how_many )
/*    p1 -- origin
      p2 -- destination
      p3 -- spare
      how_many -- number of disks to move */
{
     /* If there is only 1 disk on p1, then move it to p2 */
     /* and quit as the problem is solved.                */
     if ( how_many == 1 ) {
        printf( "\n\n\tMove top disk from peg %c to peg %c.",
               p1, p2 );
        return;
     }
     /* Otherwise:                                          */
     /*    (1) Move how_many - 1 disks from p1 to p3:       */
     /*        p1 is the origin, p3 is the destination,     */
     /*        and p2 is the spare.                         */
     /*                                                     */
     /*    (2) Move the top disk from p1 to p2.             */
     /*                                                     */
     /*    (3) Move how_many - 1 disks from p3 to p2:       */
     /*        p3 is the origin, p2 is the destination,     */
     /*        and p1 is the spare.                         */
     hanoi( p1, p3, p2, how_many - 1 );
     printf( "\n\n\tMove top disk from peg %c to peg %c.",
             p1, p2 );
     hanoi( p3, p2, p1, how_many - 1 );
}
```

Example 4.8.3: Recursive Robot Walking

```
#include <stdio.h>
#include <stdlib.h>

int walk_recur( int dist );

main()
{
     int distance,      /* how far to walk */
         count_walks;   /* how many ways to walk distance
                           meters */

     printf( "\nHow far to walk?  " );
     scanf( "%d", &distance );
     if ( distance < 1 )      /* distance must be positive */
        printf( "\n\n\tERROR -- distance must be positive." );
     else {
        count_walks = walk_recur( distance );
        printf( "\n\n\tThe robot can walk %d meters in "
                "%d ways.\n", distance, count_walks );
     }

     return EXIT_SUCCESS;
}

int  walk_recur( int dist )
{
     /* base cases: 1 meter -- 1 way, 2 meters -- 2 ways */
     if ( dist <= 2 )
        return dist;

     /* recursive call--walk( dist - 1 ) + walk( dist - 2 ) */
     return walk_recur( dist - 1 ) + walk_recur( dist - 2 );
}
```

Example 4.8.4: Nonrecursive Robot Walking

```
#include <stdio.h>
#include <stdlib.h>

int walk_non_recur( int dist );

main()
{
     int distance,       /* how far to walk */
         count_walks;    /* how many ways to walk distance
                            meters */

     printf( "\nHow far to walk?  " );
     scanf( "%d", &distance );
     if ( distance < 1 )       /* distance must be positive */
        printf( "\n\n\tERROR -- distance must be positive." );
     else {
        count_walks = walk_non_recur( distance );
        printf( "\n\n\tThe robot can walk %d meters in "
                "%d ways.\n", distance, count_walks );
     }

     return EXIT_SUCCESS;
}

/* iterative version of the walk function */

int  walk_non_recur( int dist )
{
     int  previous,       /* walk(n - 1) */
          current,        /* walk(n)     */
          temp;

     if ( dist <= 2 )
```

```
          return dist;

     previous = 1;  /* starts as walk(1) */
     current = 2;   /* starts as walk(2) */

     /* compute walk(3), then walk(4),..., until walk(dist) */
     --dist;
     while ( --dist ) {
          temp = previous;
          previous = current;
          current += temp;
     }

     return current;
}
```

Example 4.10.8: Finding the Largest Integer

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <limits.h>

int max( int how_many, ... );

main()
{
     int max_int;
     int i1, i2, i3, i4;  /* arguments to test max function */

     printf( "Enter one to four integers\n"
             "generate EOF when done\n" );

     scanf( "%d", &i1 );

     if ( scanf( "%d", &i2 ) == EOF )
          max_int = max( 1, i1 );
     else if ( scanf( "%d", &i3 ) == EOF )
          max_int = max( 2, i1, i2 );
     else if ( scanf( "%d", &i4 ) == EOF )
          max_int = max( 3, i1, i2, i3 );
     else
          max_int = max( 4, i1, i2, i3, i4 );

     printf( "\nMaximum integer = %d\n", max_int );

     return EXIT_SUCCESS;
}

/*   max -- Returns the maximum of a list of ints. The first
     argument is the number of ints over which to take the
```

```
     maximum. The ints over which to take the maximum follow
     the first argument. */

int max( int how_many, ... )
{
     int largest = INT_MIN; /* low value so first comparison
                               is correct */
     int i;
     va_list arg_addr;  /* gives location of argument after
                           how_many in max */
     int next_int;

     /* set arg_addr to the address of the first argument
        after how_many in max */
     va_start( arg_addr, how_many );

     for ( i = 0; i < how_many; i++ )
          /* va_arg returns the value of the next argument
             in max and updates arg_addr.  The second argument
             to va_arg is the data type of argument being
             read (in max). */
          if ( ( next_int = va_arg( arg_addr, int ) )
               > largest )
               largest = next_int;

     va_end( arg_addr ); /* mop up */

     return largest;
}
```

Section 5.3: Tracking and Reporting Car Sales

```
/*  This program tracks sales and inventory for a dozen brands
    of cars in a used-car business. The brands are coded as
    integers 0 through 11.

    At each day's start, the user initializes

                 cars_in_stock[ brand ]

    to total inventory for a given brand, and

                       total_cars_sold[ brand ]

    to zero. With each sale, the program updates

                       total_cars_sold[ brand ]

    by cars_sold. At day's end, the user enters a negative
    integer for a brand, which causes the program to generate
    a summary report.

*/

#include <stdio.h>
#include <stdlib.h>

/* number of car brands currently handled */
#define NUM_BRANDS 12

main()
{
       int  cars_in_stock[ NUM_BRANDS ];
       int  total_cars_sold[ NUM_BRANDS ];
       int  cars_sold;
```

```
int  brand;

/* Initialize cars_in_stock and total_cars_sold for
   each brand. */
for ( brand = 0; brand < NUM_BRANDS; ++brand ) {
   printf( "\n\n\tCars in stock for brand: %d  ",
           brand );
   scanf( "%d", &cars_in_stock[ brand ] );
   total_cars_sold[ brand ] = 0;
}

/* Record sales, updating total_cars_sold, until user
   signals halt.*/
printf( "\n\n\n" );

/* Loop until user signals halt by setting brand less
   than zero. */
 while ( brand >= 0 ) {
     printf( "\n\n\t Which brand was sold? " );
     scanf( "%d", &brand );
     /* BRAND in bounds? */
     if ( brand >= 0 && brand < NUM_BRANDS ) {
          printf( "\n\t How many cars in sale? " );
          scanf( "%d", &cars_sold );
          total_cars_sold[ brand ] += cars_sold;
     }
}

/* Generate daily report for each brand. */
printf( "\n\n\n\n\t\tDAILY REPORT" );
printf( "\n\t\t----- ------" );
for ( brand = 0; brand < NUM_BRANDS; ++brand ) {
   printf( "\n\n\t\tBrand #:\t\t\t%d", brand );
   printf( "\n\t\tInventory at day's start:\t%d",
       cars_in_stock[ brand ] );
```

```
        printf( "\n\t\tTotal sales:\t\t\t%d",
            total_cars_sold[ brand ] );
        printf( "\n\t\tInventory at day's end:\t\t%d",
            cars_in_stock[ brand ] -
            total_cars_sold[ brand ] );
        printf( "\n\t\tSales as percentage of "
            "inventory:\t\t%f",
            100 * ( (float) total_cars_sold[ brand ] /
            ( float ) cars_in_stock[ brand ] ) );
    }

    return EXIT_SUCCESS;
}
```

Section 5.8: Computing a String's Length

```
/* function to compute a string's length */
/* length expects an array of characters */
int length( char string[ ] )
{
     int  count;     /* string's length */

     for ( count = 0; string[ count ] != '\0'; ++count )
          ;
     return count;
}
```

Section 5.10: Matrix Multiplication

```
#include <stdio.h>
#include <stdlib.h>

#define MAXSIZE 20

void
  store( int m[ ][ MAXSIZE ], int n ),
  mult( int m1[ ][ MAXSIZE ], int m2[ ][ MAXSIZE ],
        int m3[ ][ MAXSIZE ], int n ),
  print( int m[ ][ MAXSIZE ], int n );

main()
{
  int n;
  int m1[ MAXSIZE ][ MAXSIZE ],
      m2[ MAXSIZE ][ MAXSIZE ],
      m3[ MAXSIZE ][ MAXSIZE ];

  /* read data into m1 and m2 and echo */
  printf( "Input matrix size: " );
  scanf( "%d", &n );
  printf( "Input first matrix by row\n" );
  store( m1, n );
  printf( "\nMatrix m1:\n" );
  print( m1, n );
  printf( "\nInput second matrix by row\n" );
  store( m2, n );
  printf( "\nMatrix m2:\n" );
  print( m2, n );

  /* multiply m1 by m2, storing product in m3 */
  mult( m1, m2, m3, n );
```

```
  /* print results */
  printf( "\nProduct m3:\n" );
  print( m3, n );

  return EXIT_SUCCESS;
}

/* Multiply matrix m1 by matrix m2, storing the
 * product in matrix m3, where all three matrices are
 * n by n in size:
 *                 n-1
 *   m3[ i ][ j ] = SUM m1[ i ][ k ] * m2[ k ][ j ]
 *                 k=0
 * and 0 <= i,j < n.
 */
void mult( int m1[ ][ MAXSIZE ],
           int m2[ ][ MAXSIZE ],
           int m3[ ][ MAXSIZE ],
           int n )
{
  int i, j, k;

  for ( i = 0; i < n; i++ )
    for ( j = 0; j < n; j++ ) {
      m3[ i ][ j ] = 0;
      for ( k = 0; k < n; k++ )
        m3[ i ][ j ] += m1[ i ][ k ] * m2[ k ][ j ];
    }
}

/* Store data in matrix by row.
 */
void store( int m[ ][ MAXSIZE ], int n )
{
  int i, j;
```

```
  for ( i = 0; i < n; i++ )
    for ( j = 0; j < n; j++ )
      scanf( "%d", &m[ i ][ j ] );
}

void print( int m[ ][ MAXSIZE ], int n )
{
  int i, j;

  for ( i = 0; i < n; i++ ) {
    for ( j = 0; j < n; j++ )
      printf( "%d ", m[ i ][ j ] );
    putchar( '\n' );
  }
}
```

Section 5.11: Sorting and Searching

```
/*   The PHONE program reads NUMB_RECS records from the file
     students.dat into the two-dimensional array students. The
     records occur one to a line, with a maximum length of
     MAX_REC_SIZE and fields formatted as follows:

         25 chars    25 chars      12 chars
       <last name> <first name> <phone number>
       /|\         /|\          /|\
        |           |            |
     Column 1    Column 26    Column 51

     MAX_REC_SIZE is thus LNAME_SIZE + FNAME_SIZE + PNUM_SIZE.
     One record occurs per line.

     PHONE sorts the records--as character strings--in
     ascending order and then does lookups. It prompts for a
     last name, padding it if necessary with blanks to
     LNAME_SIZE, and then searches the array. If the name is
     found, the entire record is written to the terminal in
     data-processing format, i.e., the fields are printed in
     their file column positions. If the name is not found, a
     message is written.

     PHONE uses selection sort for the sorting and binary
     search for the searching. The program consists of the
     functions: main, selection_sort, binary_search, and swap.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define  NUMB_RECS    30 /* number of recs in students.dat */
```

```
#define  LNAME_SIZE   25 /* first 25 chars */
#define  FNAME_SIZE   25 /* next 25 chars */
#define  PNUM_SIZE    12 /* last 12 chars */
#define  MAX_REC_SIZE (LNAME_SIZE + FNAME_SIZE + PNUM_SIZE)
#define  NO_FIND      (-999) /* flag for lookup failure */
#define  BLANK        ' '    /* used to pad candidate name so
                                that it has LNAME chars */

void selection_sort( char array[ ][ MAX_REC_SIZE + 2 ],
                     int size1 );
void swap( char array[ ][ MAX_REC_SIZE + 2 ],
           int current,
           int low );
int binary_search( char candidate[ ],
                   char array[ ][ MAX_REC_SIZE + 2 ],
                   int size );

main()
{
     /* student records (+2 is for the newline and the
        null terminator) */
     char students[ NUMB_RECS ][ MAX_REC_SIZE + 2 ];

     /* candidate in search (+2 is for the newline and the
        null terminator) */
     char candidate[ LNAME_SIZE + 2 ];

     int cand_len; /* length of candidate's name */
     int index; /* index into students */
     int i;
     FILE* fp;

     /* read the names of NUMB_RECS records into array
        students */
     fp = fopen( "students.dat", "r" );
```

```
for ( i = 0; i < NUMB_RECS; ++i )
     fgets( students[ i ], MAX_REC_SIZE + 2, fp );

/* sort the records into ascending order, by name */
selection_sort( students, NUMB_RECS );

/* Prompt the user for a student's name and look for it
   in the array students. Halt when the user signals EOF
   at the prompt. */

printf( "\n\n\Enter a student's name, "
        "or signal EOF to halt:  " );
while ( fgets( candidate,
               LNAME_SIZE + 2,
               stdin ) != NULL ) {

     /* Pad candidate with blanks to LNAME_SIZE chars. */
     /* -1 for newline stored by fgets */
     cand_len = strlen( candidate ) - 1;
     for ( i = cand_len; i < LNAME_SIZE; i++ )
             candidate[ i ] = BLANK;
     /* null terminate */
     candidate[ LNAME_SIZE ] = '\0';

     index = binary_search( candidate,
                            students,
                            NUMB_RECS );
     if ( index != NO_FIND )
          printf( "\nRecord:  %s", students[ index ] );
     else {
          /* knock off blanks */
          candidate[ cand_len ] = '\0';
          printf( "\n\t%s is not in our directory.",
                  candidate );
```

```
          }
          printf( "\n\n\Enter a student's name, "
                  "or signal EOF to halt:  " );
     }

     return EXIT_SUCCESS;
}

/*   selection_sort ( array, size1 )

     The function expects an array and its size. It sorts the
     elements into ascending order. It does not return a
     value.

     The algorithm can be sketched as follows:

     1. Repeat steps 2 and 3 for ind equals 0,1,...,size1 - 2.

     2. Select the smallest item among

      array[ ind ], array[ ind + 1 ], ..., array[ size1 - 1 ].

     3. Swap the smallest with array[ ind ].

*/

void selection_sort( char array[ ][ MAX_REC_SIZE + 2 ],
                     int size1 )
{
     int   smallest_index; /* index of smallest string seen */
     int   i, j;

     /* loop size1 - 1 times, selecting
        smallest string each time */
     for ( i = 0; i < size1 - 1; ++i ) {
```

```
        /* assume ith string is smallest */
        smallest_index = i;
        /* compare smallest against remaining strings */
        for ( j = i + 1; j < size1; ++j )
           /* look for smaller string */
           if ( strcmp( array[ j ],
                        array[ smallest_index ] ) < 0 )
             smallest_index = j;
        /* put smallest at index i */
        if ( i != smallest_index )
           swap( array, i, smallest_index );
     }
}

/* Swap array[ current ] and array[ low ]. */

void swap( char array[ ][ MAX_REC_SIZE + 2 ],
           int current,
           int low )
{
     char  temp[ MAX_REC_SIZE + 2 ];

     strcpy( temp, array[ current ] );
     strcpy( array[ current ], array[ low ] );
     strcpy( array[ low ], temp );
}

/*   binary_search( candidate, array, size )

     The function searches the sorted array

          array[ 0 ], array[ 1 ] , ..., array[ size - 1]

     for candidate, returning the appropriate index if
     candidate is found, and the NO_FIND flag otherwise.
```

```
     The algorithm can be sketched as follows:

     1. Repeat steps 2 through 4 until either the candidate is
        found or there is nothing left to search.

     2. Find the (approximate) midpoint in the array. (This is
        an index into the array.)

     3. Compare the array's midpoint item with candidate: if
        they match, return the midpoint value and halt
        (success).

     4. If the candidate is less than the midpoint item,
        search the first half of the array, discarding the
        second half; otherwise, search the second half,
        discarding the first.

     5. Return NO_FIND (failure).
*/

int binary_search( char candidate[ ],
                   char array[ ][ MAX_REC_SIZE + 2 ],
                   int size )
{
     int   len;              /* length of candidate string */
     int   first = 0;        /* first position in array */
     int   last = size - 1;  /* last position in array */
     int   mid;              /* midpoint in array */
     int   flag;             /* holds result of compare */

     len = strlen( candidate );

     /* search until list is empty */
     while ( first <= last ) {
```

```
        /* find (approximate) midpoint */
        mid = ( first + last ) / 2;
        /* Compare only len - 1 characters. - 1 is so we
           don't include the terminating newline. */
        flag = strncmp( candidate, array[ mid ], len - 1 );
        if ( flag == 0 ) /* candidate == array[ mid ] */
             return mid; /* signal success */
        if ( flag > 0 ) /* candidate > array[ mid ] */
             first = mid + 1;  /* search right half */
        else /* candidate < array[ mid ] */
             last = mid - 1;   /* search left half */
    }

    return NO_FIND;
}
```

Section 6.5: Reversing a String in Place

```
/*   This program reads a string of up to 100 characters with
     no embedded blanks and writes the reversed string. The
     function rev, invoked with the array as its single
     argument, does the work. The original and reversed
     strings occupy the same storage. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void rev( char* s );

main()
{
     char str[ 101 ]; /* storage for up to 100 chars and a
                         null terminator */

     printf( "\n\nEnter a string:\t" );
     scanf( "%s", str );
     rev( str );
     printf( "\n\nReversed string:\t%s", str );

     return EXIT_SUCCESS;
}
```

```
/*   rev expects a pointer to a character string. It reverses
     the characters in the string leaving the null terminator
     in place. */

void rev( char* s )
{
     char temp, *end;

     end = s + strlen( s ) - 1; /* end points to last nonnull
                                   character in s */
     while ( s < end ) {
          temp = *s;
          *s++ = *end;
          *end-- = temp;
     }
}
```

Section 6.10: Comparing Sorting Algorithms

```
/*   This program times various sorting functions; main
     invokes the function timer which invokes and times a
     sort. The arguments to timer are

          --the name of the algorithm
          --an array that holds the ints to sort
          --the number of items to sort
          --a pointer to the function that sorts
*/

#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define NUMB_OF_ARRAY_SIZES  8
#define MAX_SIZE_TO_SORT  5000

void fill_array( int a[ ], int count );
void selection_sort( int a[ ], int count );
void quicksort( int a[ ], int count );
void timer( char* alg_name,
            int a[ ],
            int size,
            void ( *sort_alg )( int [ ], int ) );

main()
{
     int i;   /* loop counter */
     /* sort arrays of size 100, 200, 500, 1000, 2000,
        3000, 4000, 5000 */
     int numb[ ] = { 100, 200, 500, 1000, 2000, 3000,
                     4000, 5000 };
     int a[ MAX_SIZE_TO_SORT ]; /* holds ints to sort */
```

```
     for ( i = 0; i < NUMB_OF_ARRAY_SIZES; i++ ) {
          printf( "\n\n\nSort %d items\n", numb[ i ] );
          fill_array( a, numb[ i ] );
          timer( "Selection sort", a, numb[ i ],
                 selection_sort );
          timer( "Quicksort", a, numb[ i ], quicksort );
     }

     return EXIT_SUCCESS;
}

/*   timer receives four arguments

          --the name of the algorithm
          --an array that holds the ints to sort
          --the number of items to sort
          --a pointer to the function that sorts

     timer first copies the array that holds the ints to sort
     to the local array b, so that the data passed are not
     changed. After printing the name of the sorting
     algorithm, the function is invoked and the time to sort
     is printed. The time is measured by invoking the system
     functions time, which returns the current calendar time,
     and difftime, which returns the difference in seconds
     between two calendar times.
*/

void timer( char* alg_name, /* name of algorithm */
            int a[ ],        /* data to sort */
            int size,       /* number of ints to sort */
            void ( *sort_alg )( int [ ], int ) ) /* pointer to
                                                    sort
                                                    function */
```

```
{
     int i; /* loop counter */
     int b[ MAX_SIZE_TO_SORT ]; /* array to sort */
     time_t start; /* time when sort begins */
     time_t stop;  /* time when sort ends */

     /* Copy a to b. b will be sorted. */
     for ( i = 0; i < size; i++ )
          b[ i ] = a[ i ];

     printf( "\tAlgorithm: %s\n", alg_name );
     start = time( NULL );
     /* sort b[ 0 ], ..., b[ size - 1 ] */
     ( *sort_alg )( b, size );
     stop = time( NULL );
     printf( "\t\tTime = %f seconds\n",
                  difftime( stop, start ) );
}

/*  fill_array writes random ints into

                   a[ 0 ], ..., a[ count - 1 ].

    The library function rand is used, and rand returns a
    random integer.
*/

void fill_array( int a[ ], int count )
/* a[ ] is an array of count integers */
{
     int i;

     for ( i = 0; i < count; i++ )
          a[ i ] = rand();
}
```

```
void selection_sort( int a[ ], int count )
/* a[ ] is the array to sort */
/* sort a[ 0 ], ..., a[ count - 1 ] */
{
     /* code to implement selection_sort */
}

void quicksort( int a[ ], int count )
/* a[ ] is the array to sort */
/* sort a[ 0 ], ..., a[ count - 1 ] */
{
     /* code to implement quicksort */
}
```

Section 7.5: An Interactive Calculator

```
           /**** Start of file defines.h ****/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MaxExp   (120)  /* max user input */
#define Forever    (1)
#define False      (0)
#define Stop       '!'
            /**** End of file defines.h ****/

              /**** Start of file calc.c ****/
#include "defines.h"

/* The interactive calculator simulates a hand-held
 * calculator. The calculator evaluates expressions
 * until the user signals halt by EOF. An expression
 * consists of a left operand, an operator, and a right
 * operand. The left operand is initialized to zero
 * and is automatically displayed as part of the
 * prompt for user input. The user inputs the right
 * operand, an expression that consists of constants
 * and operators. To halt, the user enters the stop
 * character ! as the first operator in the right operand.
 */

/* Left-hand operands of expression
 * to be calculated. Left-hand side is the target
 * as well as one of the sources.
 */
int lhs = 0;

/* input/output buffer -- + 2 for newline and null */
```

```
char buffer[ MaxExp + 2 ];

/* legal operators and stop flag */
static char* ops = "+-*/%!";

/* function declarations */
void prompt( void ), input( void ), output( void ),
     update_lhs( void );
int bad_op( char );

main()
{
  while ( Forever ) {
    prompt();
    input();
    output();
  }
}

int bad_op( char c )
{
  return NULL == strchr( ops, c );
}

/* Parse the right operand, an expression that
 * consists of binary operators and integer
 * constants. In case of error, do not update
 * left operand. Otherwise, set left operand
 * to current value of calculation.
 */
void update_lhs()
{
  char *next = buffer, *last;
  char op[ 10 ], numStr[ 20 ];
  int num;
```

```
  int temp = lhs; /* copy lhs in case of error */

  /* Loop until right operand parsed. */
  last = next + strlen( buffer );
  while ( next < last ) {
    /* read next operator and integer constant */
    if ( sscanf( next, "%s %s", op, numStr ) < 2 ||
         strlen( op ) != 1                       ||
         bad_op( op[ 0 ] ) )
      return;

    /* Exit? */
    if ( Stop == op[ 0 ] ) {
      printf( " **** End of Calculation ****\n\n" );
      exit( EXIT_SUCCESS );
    }

    /* convert numeric string to integer */
    num = atoi( numStr );

    switch ( op[ 0 ] ) {
    case '+': temp += num; break;
    case '-': temp -= num; break;
    case '*': temp *= num; break;
    case '/': temp /= num; break;
    case '%': temp %= num; break;
    }
    /* reset next -- + 2 for blanks that separate
     * operators and integer constants.
     */
    next += strlen( op ) + strlen( numStr ) + 2;
  }
  lhs = temp; /* new current value */
}
             /**** End of file calc.c ****/
```

```
               /**** Start of file io.c ****/
#include "defines.h"

void update_lhs( void );

/** extern variable declarations **/
extern char buffer[ MaxExp + 2 ];
extern int lhs;

void prompt( void )
{
  printf( "\n%d ", lhs );
}

void input( void )
{
  char* nl;

  fgets( buffer, MaxExp, stdin );
  /* eliminate newline */
  if ( ( nl = strrchr( buffer, '\n' ) ) != NULL )
    *nl = '\0';
}

void output( void )
{
  printf( "%d %s == ", lhs, buffer );
  update_lhs();
  printf( "%d\n", lhs );
}

                /**** End of file io.c ****/
```

Example 7.8.11: Sorting Using `qsort`

```
#include <stdio.h>
#include <stdlib.h>

int comp( const void* e1, const void* e2 );

main()
{
  int arr[ ] = /* array to sort*/
    { 7, 4, 3, 9, 2, 6, 1, 5, 8, 0 };
  int n = 10; /* number of elements in array */
  int i;
  /* sort the array */
  qsort( arr, n, sizeof ( arr[ 0 ] ), comp );
  /* print in sorted order */
  for ( i = 0; i < n; i++ )
    printf( "%d ", arr[ i ] );
  printf( "\n" );

  return EXIT_SUCCESS;
}

int comp( const void* e1, const void* e2 )
{
  return *( ( int* ) e1 ) - *( ( int* ) e2 );
}
```

Section 8.8: A Random Access File

```
          /***** RANDOM ACCESS FILE PROGRAM *****/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define  FILE_SIZE    419   /* slots 0 through 418 */
#define  RECORD_SIZE  47    /* 46 + status flag */
#define  KEY_SIZE      9    /* 9-digit unique identifier */
#define  TOTAL_BYTES   ( RECORD_SIZE * FILE_SIZE )
#define  INBUFF_SIZE  81    /* size of user buffer */
#define  TAKEN         'T'  /* slot that holds a record */
#define  FREE          'F'  /* slot that never held record */
#define  DELETED       'D'  /* slot that holds del record */

const int FOUND = 1;
const int NOT_FOUND = 0;
const char BLANK = ' ';
const int DIVISOR = 419;  /* used in hashing */

FILE* fptr;
const char file[ ] = "records.dat";
char inbuff[ INBUFF_SIZE ];  /* buffer for user input */
int    count;                /* count of records in file */

void add_record( void ), find_record( void ),
     remove_record( void ), initialize( void ),
     set_count( void );
long get_address( const char* key );
int locate( const char* key ), print_menu( void );
```

```
main()
{
     int choice;

     /* open the file "records.dat" and count the number of
        records it contains, if it exists; otherwise, create
        and initialize it  */
     if ( ( fptr = fopen( file, "r+b" ) ) != NULL )
          set_count();
     else {
          fptr = fopen( file, "w+b" );
          initialize();
     }

     /* Do file operations until user signals halt. */
     while ( ( choice = print_menu() ) != 0 )
          switch ( choice ) {
          case 1:
               add_record();
               break;
          case 2:
               find_record();
               break;
          case 3:
               remove_record();
               break;
          default:
               printf( "\n\n\t\t\tIllegal choice. "
                       "Please choose again.\n" );
               break;
          }
     fclose( fptr );

     return EXIT_SUCCESS;
}
```

```
/*   Create a relative file and indicate that FILE_SIZE record
     slots are open in it. */
void  initialize( void )
{
     int  i;
     char dummy[ RECORD_SIZE + 1 ];

     /* Create a dummy record to store repeatedly in file. */
     dummy[ 0 ] = FREE;       /* status unoccupied */
     for ( i = 1; i < RECORD_SIZE; ++i )
         dummy[ i ] = BLANK;
     dummy[ RECORD_SIZE ] = '\0';
     for ( i = 0; i < FILE_SIZE; ++i )
         fputs( dummy, fptr );
     count = 0;
}

/* Count occupied slots in file. */
void  set_count( void )
{
     int  i;

     /* For fseek:

            SEEK_SET means seek from beginning of file
            SEEK_CUR means seek from current file marker
            SEEK_END means seek from end of file

        All three are defined in stdio.h.
     */

     fseek( fptr, 0, SEEK_SET );

     for ( i = 0; i < FILE_SIZE; ++i ) {
```

```
        if ( fgetc( fptr ) == TAKEN )
           ++count;
        fseek( fptr, RECORD_SIZE - 1, SEEK_CUR );
    }
}

/* Print a menu of choices and return whichever the user
   picks. */
int  print_menu( void )
{
    int  choice;

    printf( "\n\n\n\t\t\t"
            "1   -- add a record\n\t\t\t"
            "2   -- find a record\n\t\t\t"
            "3   -- remove a record\n\t\t\t"
            "0   -- exit the program\n\n\t\t\t"
            "Please pick a number:   " );
    fgets( inbuff, INBUFF_SIZE, stdin );
    sscanf( inbuff, "%d", &choice );
    return choice;
}

/* Add a record to the file */
void  add_record( void )
{
    char  record[ RECORD_SIZE + 1 ];

    if ( count >= FILE_SIZE ) {  /* full file? */
        printf( "\a\n\n\n\nFile is full.\n" );
    }
    else {
        do {
           printf( "\n\nPlease enter new record's key:  " );
           fgets( inbuff, INBUFF_SIZE, stdin );
```

```
            /* input must be KEY_SIZE + newline */
            if ( strlen( inbuff ) != KEY_SIZE + 1 )
                 printf( "\aKey size must be %d", KEY_SIZE );
         } while ( strlen( inbuff ) != KEY_SIZE + 1 );
         if ( locate( inbuff ) == FOUND ) {
            printf( "\aThere is already a record with "
                    "the key: %s\n", inbuff );
            return;
         }

         /* copy key */
         strncpy( &record[ 1 ], inbuff, KEY_SIZE );
         do {
            printf( "\nPlease enter data: " );
            fgets( inbuff, INBUFF_SIZE, stdin );
            /* input must be less than or equal to
               RECORD_SIZE - KEY_SIZE - 1 counting
               the newline */
            if ( strlen( inbuff )
                 > RECORD_SIZE - KEY_SIZE - 1 )
                 printf( "\aRecord size must be less than"
                         " or equal to %d"
                         " counting the newline\n",
                         RECORD_SIZE - KEY_SIZE - 1 );
         } while ( strlen( inbuff )
                   > RECORD_SIZE - KEY_SIZE - 1 );
         strcpy( &record[ 1 + KEY_SIZE ], inbuff );
         record[ 0 ] = TAKEN;   /* mark status as taken */
         fputs( record, fptr );  /* store record */
         ++count;
    }
}
```

```
/* Return address to which a key hashes. */
long  get_address( const char* key )
{
     long int_key, address;

     int_key = atol( key );  /* convert */
     address = ( int_key % DIVISOR ) * RECORD_SIZE; /* hash */
     return address;
}

/* Remove a record from the file by marking its slot DELETED.
*/
void  remove_record( void )
{
     char key[ KEY_SIZE + 1 ];

     printf( "\n\nPlease enter record's key:  " );
     fgets( inbuff, INBUFF_SIZE, stdin );
     strncpy( key, inbuff, KEY_SIZE );
     key[ KEY_SIZE ] = '\0';
     if ( locate( key ) != FOUND )
          printf( "\a\nThere is no record with the key: %s\n",
                  key );
     else {
          --count;
          fputc( DELETED, fptr );
          printf( "\nRecord has been deleted.\n" );
     }
}

/* Find a record with a given key and print it. */
void  find_record( void )
{
     char key[ KEY_SIZE + 1 ], record[ RECORD_SIZE + 1 ];
```

```
     printf( "\n\nPlease enter record's key:   " );
     fgets( inbuff, INBUFF_SIZE, stdin );
     strncpy( key, inbuff, KEY_SIZE );
     key[ KEY_SIZE ] = '\0';
     if ( locate( key ) == NOT_FOUND )
          printf( "\aThere is no record with the key: %s",
                  key );
     else {
          /* read it */
          fgets( record, RECORD_SIZE + 1, fptr );
          printf( "Record:  %s", record + KEY_SIZE + 1 );
     }
}

/* locate searches for the record with given key. locate
   returns 1 if the record is found and 0 if the record is not
   found. Furthermore, if the record is found, locate sets the
   file position marker to the found record. If the record is
   not found, locate sets the file position marker to the
   unoccupied slot (DELETED or FREE), if any, to which key
   hashes, using linear probing to find such a slot if
   necessary. If the record is not found and the file is full,
   locate sets the file position marker to the hash address of
   key.  */

int locate( const char* key )
{
     long address,       /* current offset in file */
          start_address, /* initial (hash) offset in file */
          unocc_address; /* first DELETED offset in file, if
                            any; otherwise, equal to
                            start_address */
     int  delete_flag = 0; /* 0 if no DELETED slot found;
                              1 otherwise */
     char stored_key[ KEY_SIZE + 1 ];
```

```
address = get_address( key );  /* hash to address */
unocc_address = start_address = address;

do {
     fseek( fptr, address, SEEK_SET );  /* find slot */
     switch ( fgetc( fptr ) ) {  /* check status */
     case DELETED:
          /* if first visit to a DELETED slot,
             mark its location and set flag */
          if ( !delete_flag ) {
               unocc_address = address;
               delete_flag = 1;
          }
          break;
     case FREE:
          /* end of search. if first unoccupied slot
             is a DELETED slot, reset file position
             marker to that slot; otherwise, reset
             file position marker to present FREE slot
             which is the first unoccupied slot.  */
          if ( delete_flag )
               fseek( fptr, unocc_address, SEEK_SET );
          else
               fseek( fptr, address, SEEK_SET );
          return NOT_FOUND;
     case TAKEN:
          /* extract key */
          fseek( fptr, address + 1, SEEK_SET );
          fgets( stored_key, KEY_SIZE + 1, fptr );
          /* key equals stored_key? */
          if ( strncmp( key, stored_key,
                        KEY_SIZE ) == 0 ) {
               /* restore file position marker */
               fseek( fptr, address, SEEK_SET );
```

```
                return FOUND;
            }
            break;
        }
        /* probe */
        address = ( address + RECORD_SIZE ) % TOTAL_BYTES;
    } while ( address != start_address );

    /* reset file position marker to first DELETED slot or
       to initial slot */
    fseek( fptr, unocc_address, SEEK_SET );
    return NOT_FOUND;
}
```

Section 9.7: Sales Reporting

```
***** BEGIN sales.c *****/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sales.h"

/** arrays **/

/* sales by account */
Sales salesTbl[ MaxSales ];

/* pointers to salesTbl */
Sales* salesPtr[ MaxSales ];

/* sales personnel */
SalesPer salesPerTbl[ MaxSalesPer ];

/* pointers to salesPerTbl */
SalesPer* salesPerPtr[ MaxSalesPer ];

/** array counts **/
int salesCnt, salesPerCnt;

/** function declarations **/
void
  readSales( const char* infile ),
  readSalesPer( const char* infile ),
  quicksort( void* array,
             size_t count,
             size_t size,
             int ( *comp )( const void* e1,
                            const void* e2 ) ),
  report( void );
```

```
void*
  findPivot( const void* array,
             size_t count,
             size_t size,
             int ( *comp )( const void* e1,
                            const void* e2 ) );
int
  partition( void* array,
             size_t count,
             size_t size,
             int ( *comp )( const void* e1, const void* e2 ),
             void* pivot ),
  compAccId( const void* e1, const void* e2 ),
  compName( const void* e1, const void* e2 );
FILE*
  openOrExit( const char* file, const char* mode );

main()
{
  /* read sales data and sort */
  readSales( SalesFile );
  quicksort( salesPtr,
             salesCnt,
             sizeof ( salesPtr[ 0 ] ),
             compAccId );

  /* read salesperson data and sort */
  readSalesPer( SalesPerFile );
  quicksort( salesPerPtr,
             salesPerCnt,
             sizeof ( salesPerPtr[ 0 ] ),
             compName );

  /* print sales report */
  report();
```

```
  return EXIT_SUCCESS;
}

/* Read data into array salesTbl and set SalesCnt */
void readSales( const char* infile )
{
  FILE* fptr;
  int i = 0;
  char buffer[ MaxBuffer + 2 ]; /* input buffer */

  fptr = openOrExit( infile, "rb" );
  salesCnt = 0;

  while ( fgets( buffer, MaxBuffer, fptr ) ) {
    sscanf( buffer, "%d %s %f %f",
           &salesTbl[ i ].accId,
            salesTbl[ i ].accName,
           &salesTbl[ i ].act,
           &salesTbl[ i ].proj );
    i++;
  }
  fclose( fptr );
  salesCnt = i;

  /* Set pointers in salesPtr to addresses of
   * corresponding elements in salesTbl.
   */
  for ( i = 0; i < salesCnt; i++ )
    salesPtr[ i ] = &salesTbl[ i ];
}

/* Read data into array salesPerTbl and set SalesPerCnt */
void readSalesPer( const char* infile )
{
```

```
FILE* fptr;
int i = 0, j;
char* del;
char id[ MaxName + 1 ];
char buffer[ MaxBuffer + 2 ];  /* input buffer */
Sales temp, *tempPtr = &temp, **ret;

fptr = openOrExit( infile, "rb" );
salesPerCnt = 0;

while ( fgets( buffer, MaxBuffer, fptr ) ) {
  /* Read basic fields. */
  sscanf( buffer, "%s %s %s %f %f",
          salesPerTbl[ i ].name,
          salesPerTbl[ i ].boss,
          salesPerTbl[ i ].reg,
         &salesPerTbl[ i ].base,
         &salesPerTbl[ i ].bonus );

  /*** read account ids and set
       pointers to Sales structures ***/
  salesPerTbl[ i ].accCnt = 0;
  j = 0;
  /* find accounts delimiter within buffer */
  del = strrchr( buffer, AccDel );
  del++;
  while ( *del && sscanf( del, "%s", id ) != EOF ) {
    tempPtr -> accId = atoi( id );
    del += strlen( id ) + 1; /* move beyond current id */
    /* Search for sales record by account id. */
    if ( ret = bsearch( &tempPtr,
                        salesPtr,
                        salesCnt,
                        sizeof ( salesPtr[ 0 ] ),
                        compAccId ) ) {
```

```
        salesPerTbl[ i ].accCnt++;
        salesPerTbl[ i ].accs[ j++ ] = *ret;
      }
    }
    i++;
  }
  fclose( fptr );
  salesPerCnt = i;

  /* Copy address of sales table entries
   * into pointer array.
   */
  for ( i = 0; i < salesPerCnt; i++ )
    salesPerPtr[ i ] = &salesPerTbl[ i ];
}

/* Open file in specified mode, returning
 * file pointer if successful; otherwise,
 * issue warning message and exit program.
 */
FILE* openOrExit( const char* file, const char* mode )
{
  FILE* fptr;
  if ( NULL == ( fptr = fopen( file, mode ) ) ) {
    fprintf( stderr,
             "!! Can't open %s in mode %s. Exiting. !!\n",
             file, mode );
    exit( EXIT_SUCCESS );
  }
  return fptr;
}

void quicksort( void* array,
                size_t count,
                size_t size,
```

```
                int ( *comp )( const void* e1,
                               const void* e2 ) )
{
  void* pivot;
  int i;

  /* If array not yet sorted:
   *   (1) find a pivot
   *   (2) partition into two subarrays, left and right
   *   (3) quicksort the left subarray
   *   (4) quicksort the right subarray
   */
  if ( pivot = findPivot( array, count, size, comp ) ) {
    i = partition( array, count, size, comp, pivot );
    quicksort( array, i, size, comp );
    quicksort( ( char* ) array + i * size, count - i,
               size, comp );
  }
}

/* Return the larger of the first element and any
 * other that differs from it. If all elements the
 * same, return NULL.
 */
void* findPivot( const void* array,
                 size_t count,
                 size_t size,
                 int ( *comp )( const void* e1,
                                const void* e2 ) )
{
  int i = 1, flag;
  void* next;

  while ( i < count ) {
    next = ( char* ) array + i * size;
```

```
    flag = comp( array, next );
    if ( flag > 0 )          /* array[ 0 ] > array[ i ] */
      return array;
    else if ( flag < 0 )     /* array[ i ] > array[ 0 ] */
      return next;
    else                     /* array[ 0 ] == array[ i ] */
      i++;
  }
  return NULL;  /* all elements the same */
}

/* Partition an array into two subarrays, the first
 * holding elements <= pivot and the second holding
 * elements > pivot.
 */
int partition( void* array,
               size_t count,
               size_t size,
               int ( *comp )( const void* e1,
                              const void* e2 ),
               void* pivot )
{
  int left = 0, right = count - 1, n = count;
  void* next;
  char temp[ MaxBuffer ];

  if ( size >= MaxBuffer ) {
    fprintf( stderr,
            "\n!! MaxBuffer too small. Exiting. !!\n\n" );
    exit( EXIT_SUCCESS );
  }

  while ( left <= right ) {

    /* Search from left to right for 1st element >= pivot. */
```

```
    next = ( char* ) array + left * size;
    while ( left < n && comp( next, pivot ) < 0 ) {
      left++;
      next = ( char* ) array + left * size;
    }

    /* Search from right to left for 1st element < pivot. */
    next = ( char* ) array + right * size;
    while ( right >= 0 && comp( next, pivot ) >= 0 ) {
      right--;
      next = ( char* ) array + right * size;
    }

    /* Swap current left and right elements to put them in
     * relative order with respect to each other and pivot.
     */
    if ( left < right ) {
      char *p1 = ( char* ) array + left * size,
           *p2 = ( char* ) array + right * size;
      memcpy( temp, p1, size );
      memcpy( p1, p2, size );
      memcpy( p2, temp, size );
      left++;
      right--;
    }
  }
  return left;
}

int compAccId( const void* e1, const void* e2 )
{
  Sales
    *t1 = *( ( Sales** ) e1 ),
    *t2 = *( ( Sales** )  e2 );
```

```
  return t1 -> accId - t2 -> accId;
}

int compName( const void* e1, const void* e2 )
{
  SalesPer
    *t1 = *( ( SalesPer** ) e1 ),
    *t2 = *( ( SalesPer** ) e2 );
  return strcmp( t1 -> name, t2 -> name );
}

void report( void )
{
  int i, j;

  printf( "\nSales report by salesperson:\n\n" );
  for ( i = 0; i < salesPerCnt; i++ ) {
    printf( "\n\tSalesperson:  %s\n",
            salesPerPtr[ i ] -> name );
    printf( "\tSales coach:  %s\n",
            salesPerPtr[ i ] -> boss );
    printf( "\tSales region: %s\n",
            salesPerPtr[ i ] -> reg );
    printf( "\tBase salary and bonus:  %.2f  %.2f\n",
            salesPerPtr[ i ] -> base,
            salesPerPtr[ i ] -> bonus );
    printf( "\t%d Account(s): \n",
            salesPerPtr[ i ] -> accCnt );
    for ( j = 0; j < salesPerPtr[ i ] -> accCnt; j++ ) {
      printf( "\n" );
      printf( "\t\tName:       %s\n",
              salesPerPtr[ i ] -> accs[ j ] -> accName );
      printf( "\t\tId:         %d\n",
              salesPerPtr[ i ] -> accs[ j ] -> accId );
      printf( "\t\tProjected:  %.2f\n",
```

```
              salesPerPtr[ i ] -> accs[ j ] -> proj );
      printf( "\t\tActual:     %.2f\n",
              salesPerPtr[ i ] -> accs[ j ] -> act );
    }
  }
}
/***** END sales.c *****/
```

Example 10.2.3: Generating a Linked List

```
#include <stdio.h>
#include <stdlib.h>

/* declare a self-referential structure */
typedef  struct elephant {
     char             name[ 10 ];
     struct elephant* next;
} ELEPHANT;

void      print_elephants( const ELEPHANT* ptr );
ELEPHANT  *get_elephants( void ), *start;

main()
{
     start = get_elephants();
     print_elephants( start );

     return EXIT_SUCCESS;
}

/*   get_elephants allocates run-time storage for nodes. It
     builds the linked list and stores user-supplied names in
     the name fields of the nodes. It returns a pointer to the
     first such node.
*/

ELEPHANT* get_elephants( void )
{
     ELEPHANT  *current, *first;
     int response;

     /* allocate first node */
     current = first = malloc( sizeof ( ELEPHANT ) );
```

```
    /* store name of first elephant */
    printf( "\n\n\tNAME:\t" );
    scanf( "%s", current -> name );

    /* prompt user about another elephant */
    printf( "\n\n\n\tAdd another? (1 == yes, 0 == no)\t" );
    scanf( "%d", &response );

    /* Add elephants to list until user signals halt. */
    while ( response ) {
         /* try to allocate another elephant node */
         if ( ( current -> next =
              malloc( sizeof ( ELEPHANT ) ) ) ==
              NULL ) {
              printf( "Out of memory\n"
                      "Can't add more elephants\n" );
              return first;
         }
         current = current -> next;

         /* store name of next elephant */
         printf( "\n\n\tNAME:\t" );
         scanf( "%s", current -> name );

         /* prompt user about another elephant */
         printf( "\n\n\n\tAdd another? "
                 "(1 == yes, 0 == no)\t" );
         scanf( "%d", &response );
    }

    /* set link field in last node to NULL */
    current -> next = NULL;

    return first;
}
```

```
/*   print_elephants steps through the linked list pointed to
     by ptr and prints the name field in each node as well as
     the position of the node in the list
*/

void  print_elephants( const ELEPHANT* ptr )
{
     int count = 1;

     printf( "\n\n\n" );
     while ( ptr !=  NULL ) {
          printf( "\nElephant number %d is %s.",
               count++, ptr -> name );
          ptr = ptr -> next;
     }
}
```

Example 10.4.2: A Stack

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 100

typedef struct tray {
   char  color[ 10 ];    /* its color */
   int   id;             /* its unique id number */
   struct tray* below;   /* pointer to successor on stack */
} TRAY;

TRAY* top = NULL;    /* pointer to top TRAY on stack */
int   currsize = 0;  /* number of items on stack */

void  get_data( TRAY* ptr ), put_data( const TRAY* ptr );
TRAY *pop( void ), *push( TRAY tr );

main()
{
     int   ans, flag;
     TRAY t, *ptr;

     /* do stack operations until user signals halt */
     do {
          do {
               printf( "\n\n\tEnter 1 to push, 2 to pop:  " );
               scanf( "%d", &ans );
               printf( "\n" );
               switch ( ans ) {
               case 1: /* get a TRAY and add it to stack */
                    get_data( &t );
                    if ( push( t ) == NULL )
                         printf( "\n\nSTACK FULL\n\n" );
```

```
                    break;
               case 2: /* delete TRAY from stack and print */
                    if ( ( ptr = pop() ) != NULL )
                         put_data( ptr );
                    else
                         printf( "\n\nSTACK EMPTY\n\n" );
                    break;
               default:
                    printf( "\nIllegal response\n" );
                    break;
               }
          } while ( ans != 1 && ans != 2 );
          printf( "\n\n\n1 to continue, 0 to quit: " );
          scanf( "%d", &flag );
          printf( "\n" );
     } while ( flag );

     return EXIT_SUCCESS;
}

/*   get_data prompts the user for a TRAY's color and id and
     stores it at the address passed.
*/

void  get_data( TRAY* ptr )
{
     printf( "\nenter the tray's color: " );
     scanf( "%s", ptr -> color );
     printf( "\nenter the tray's id: " );
     scanf( "%d", &( ptr -> id ) );
     printf( "\n" );
}

/*   put_data writes the color and id of the TRAY whose
address is passed.
```

```
*/

void put_data( const TRAY* ptr )
{
     printf( "\ntray's color: %s\n", ptr -> color );
     printf( "\ntray's id: %d\n", ptr -> id );
}

/*   If the stack is full, push returns NULL. Otherwise, push
     allocates storage for a TRAY, copies the data passed into
     the allocated storage, adds the node to the linked list,
     updates top and the current size of the stack, and
     returns the address of the TRAY added.
*/

TRAY* push( TRAY tr )
{
     TRAY* ptr;

     if ( currsize  >=  SIZE ) /* stack full? */
          return NULL;
     ptr = malloc( sizeof ( TRAY ) ); /* new TRAY */
     *ptr = tr; /* store data */
     ptr -> below = top; /* push it on stack */
     top = ptr; /* update top */
     ++currsize; /* update current stack size */

     return ptr;
}

/*   If the stack is empty, pop returns NULL. Otherwise, pop
     copies the top TRAY to permanent storage, updates top,
     frees the stack storage, updates the current size of the
     stack, and returns the address of the TRAY.
*/
```

```
TRAY* pop( void )
{
     static TRAY popped_tray;
     TRAY* ptr;

     if ( currsize < 1 ) /* empty stack? */
          return NULL;
     popped_tray = *top; /* copy data to return */
     /* save address of 1st node for garbage collection */
     ptr = top;
     top = top -> below; /* update top */
     free( ptr ); /* collect garbage */
     --currsize; /* update current size */

     return &popped_tray;
}
```

Example 10.4.4: A Queue

```
#include <stdio.h>
#include <stdlib.h>

#define  SIZE  100

typedef  struct customer {
   char    lname[ 25 ];    /* last name */
   char    fname[ 15 ];    /* first name */
   int     account_no;     /* account number */
   float   balance;        /* balance */
   struct  customer* succ; /* successor on the queue */
} CUSTOMER;

CUSTOMER  *front, *rear; /* exit & entry positions in queue */
int       count = 0;     /* count of items in queue */

void  get_data( CUSTOMER* ptr ),
      put_data( const CUSTOMER* ptr );
CUSTOMER *insert( CUSTOMER cust ), *remov( void );

main()
{
     int   ans, flag;
     CUSTOMER cu, *ptr;

     /* do queue operations until user signals halt */
     do {
          do {
               printf( "\n\n\tEnter 1 to insert, "
                       "2 to remove:  " );
               scanf( "%d", &ans );
               printf( "\n" );
               switch ( ans ) {
```

```
               case 1: /* get a CUSTOMER and add to queue */
                    get_data( &cu );
                    if ( insert( cu ) == NULL )
                         printf( "\n\nQUEUE FULL\n\n" );
                    break;
               case 2: /* del CUSTOMER from queue & print */
                    if ( ( ptr = remov() ) != NULL )
                         put_data( ptr );
                    else
                         printf( "\n\nQUEUE EMPTY\n\n" );
                    break;
               default:
                    printf( "\nIllegal response\n" );
                    break;
               }
          } while ( ans != 1 && ans != 2 );
          printf( "\n\n\n1 to continue, 0 to quit: " );
          scanf( "%d", &flag );
          printf( "\n" );
     } while ( flag );

     return EXIT_SUCCESS;
}

/*   get_data prompts the user for a CUSTOMER's last name,
     first name, account_no, and balance stores the data at
     the address passed.
*/

void  get_data( CUSTOMER* ptr )
{
     printf( "\nenter the customer's last name: " );
     scanf( "%s", ptr -> lname );
     printf( "\nenter the customer's first name: " );
     scanf( "%s", ptr -> fname );
```

```
     printf( "\nenter the customer's account number: " );
     scanf( "%d", &( ptr -> account_no ) );
     printf( "\nenter the customer's balance: " );
     scanf( "%f", &( ptr -> balance ) );
     printf( "\n" );
}

/*   put_data writes the last name, first name, account_no,
     and balance of the CUSTOMER whose address is passed.
*/

void put_data( const CUSTOMER* ptr )
{
     printf( "\ncustomer's name: %s, %s\n", ptr -> lname,
             ptr -> fname  );
     printf( "\ncustomer's account number: %d\n",
             ptr -> account_no );
     printf( "\ncustomer's balance: %f\n", ptr -> balance );
}

/*   If the queue is full, insert returns NULL. Otherwise,
     insert allocates storage for a CUSTOMER, copies the data
     passed into the allocated storage, adds the node to the
     rear (last node in the linked list), updates rear, NULLs
     the link field of the new node, updates count, and
     returns the address of the CUSTOMER added.
*/

CUSTOMER* insert( CUSTOMER cust )
{
     CUSTOMER* ptr;

     if ( count  >=  SIZE ) /* queue full? */
          return NULL;
     ptr = malloc( sizeof ( CUSTOMER ) ); /* new CUSTOMER */
```

```
     *ptr = cust; /* store data */

     if ( count == 0 ) /* empty queue? */
          front = ptr; /* front points to 1st node in list */
     else
          /* if queue not empty, add at end */
          rear -> succ = ptr;
     rear = ptr; /* update rear */
     ptr -> succ = NULL; /* NULL last succ field */
     ++count; /* update count */

     return ptr;
}

/*   If the queue is empty, remov returns NULL. Otherwise,
     remov copies the CUSTOMER at the front (first node in the
     linked list) to permanent storage, updates front, frees
     the node, updates count, and returns the address of the
     CUSTOMER. */
CUSTOMER* remov( void )
{
     static CUSTOMER removed_cust;
     CUSTOMER* next;
     if ( count == 0 ) /* empty queue? */
          return NULL;
     removed_cust = *front; /* copy CUSTOMER at front */
     next = front; /* save front node's address for freeing */
     front = front -> succ; /* remove front node */
     free( next ); /* deallocate storage */
     --count; /* update count */

     return &removed_cust;
}
```

Section 10.9: Heuristic Graph Search

```
#include <stdio.h>
#include <stdlib.h>

#define  SIZE   19      /* number of cities */

int      START;         /* index of start city,
                           set interactively */

#define  GOAL   17      /* index of goal city, Vienna */

#define  CLOSED  0      /* flags for open and closed lists */
#define  OPEN    1

#define  TRUE    1
#define  FALSE   0

typedef  struct city {
         char           name[ 15 ];
         int            ind;   /* position in array cities */
         int            g_val; /* optimal path to city */
         int            h_est; /* estimate to goal */
         int            f_est; /* g_value + h_estimate */
         int            open;  /* open list */
         int            closed;  /* closed list */
         struct city*   adjcs[ SIZE ]; /* adjacent cities */
         struct city*   backptr;  /* optimal path pointer */
} CITY;

CITY  cities[ SIZE ];

/* distances between two cities */
int   distances[ SIZE ][ SIZE ];
```

```
int   open_count = 0;   /* number of items on open list */

void  get_input( void ),
      set_adjacency_lists( void ),
      astar( void );
void  failure( void ), success( const CITY* ptr ),
      expand( int index );
int   get_best_prospect( void );
void  update_values( CITY* ptr ),
      update_path( CITY* ptr, int index );
void  get_start_city( void );
void  print_cities( const CITY* ptr );

main()
{
        get_input();
        set_adjacency_lists();

        cities[ START ].backptr = NULL; /* path starts here */
        cities[ START ].g_val = 0;

        /* put START on open list */
        cities[ START ].open = TRUE;
        ++open_count;

        astar();  /* search */

        return EXIT_SUCCESS;
}

void astar( void )
{
        int best;

        if ( open_count == 0 )  /* any more options? */
```

```
            failure(); /* if not, fail */
        else {
            /* get most promising city */
            best = get_best_prospect();
            if ( best == GOAL ) /* goal city? */
                /* if so, succeed */
                success( &cities[ GOAL ] );
            else { /* otherwise: */
                /* remove from open */
                cities[ best ].open = FALSE;
                --open_count;
                /* put on closed */
                cities[ best ].closed = TRUE;
                expand( best ); /* expand it */
                astar();  /* keep searching */
            }
        }
}

int get_best_prospect( void )
{
        int  i = 0, best;

        /* Find 1st city in open list. */
        while ( cities[ i++ ].open == FALSE )
                        ;
        /* Assume it's the best. */
        best = --i;

        /* Look for something better. */
        for ( i = best + 1; i < SIZE; ++i )
            if ( cities[ i ].open == TRUE  &&
                 cities[ i ].f_est  <  cities[ best ].f_est )
               best = i;
```

```
        return best;
}

void expand( int index )
{
        int i = 0;

        while ( cities[ index ].adjcs[ i ] != NULL ) {

             /* If adjacent city is not on OPEN or CLOSED,
                then (1) set its backpointer to current city,
                (2) compute its g_val and f_est, and (3) put
                it on OPEN.         */
             if ( cities[ index ].adjcs[ i ]
                  -> open == FALSE  &&
                  cities[ index ].adjcs[ i ]
                  -> closed == FALSE ) {
                cities[ index ].adjcs[ i ] -> backptr =
                   &cities[ index ];
                update_values( cities[ index ].adjcs[ i ] );
                cities[ index ].adjcs[ i ] -> open = TRUE;
                ++open_count;
             }
             /* Otherwise, see if its backpointer needs
                changing. */
             else
                update_path( cities[ index ].adjcs[ i ],
                             index );
             ++i;
        }
}

void  update_values( CITY* ptr )
{
        int ind1, ind2;
```

```
        ind1 = ptr -> backptr -> ind;   /* current city */
        ind2 = ptr -> ind;              /* predecessor */
        ptr -> g_val = ptr -> backptr -> g_val
                     +  distances[ ind1 ][ ind2 ];
        ptr -> f_est = ptr -> g_val  +  ptr -> h_est;
}

void  update_path( CITY* ptr, int index )
{
        CITY* best;

        /* Assume current path is best. */
        best = ptr -> backptr;

        /* Compare with new path. */
        if ( best != NULL
             && cities[ index ].g_val  <  best -> g_val ) {
           /* If new path is better than the old one,
              then reset backpointer and update g_val and
              f_est. */
           ptr -> backptr = &cities[ index ];
           update_values( ptr );

           /* Is city on CLOSED? */
           if ( ptr -> closed == TRUE ) {
              /* If so, put back on OPEN. */
              ptr -> closed = FALSE;
              ptr -> open = TRUE;
              ++open_count;
           }
        }
}
```

```
void get_input( void )
{
        FILE* fptr;
        int   i, j;

        /* Read cities' names into structure variables. */
        fptr = fopen( "names.dat", "r" );
        for ( i = 0; i < SIZE; ++i ) {
             fscanf( fptr, "%s%d",
                     cities[ i ].name, &cities[ i ].h_est );
             cities[ i ].ind = i;
             cities[ i ].open = FALSE;
             cities[ i ].closed = FALSE;
        }
        fclose( fptr );

        /* Read distances between cities into matrix. */
        fptr = fopen( "distance.dat", "r" );
        for ( i = 0; i < SIZE; ++i )
             for ( j = 0; j < SIZE; ++j )
                  fscanf( fptr, "%d", &distances[ i ][ j ] );
        fclose( fptr );

        /* Prompt user for START city, and read its index into
           START. */
        get_start_city();
}

void get_start_city( void )
{
        printf( "\n\n\t\tPlease select a start city." );
        printf( "\n\t\tEnter a number associated with one "
                "of the cities.\n" );

        printf( "\n\t\tAmsterdam  ---  1\tLisbon  --- 10" );
```

```
        printf( "\n\t\tBelgrade   ---  2\tMadrid  --- 11" );
        printf( "\n\t\tBerlin     ---  3\tMunich  --- 12" );
        printf( "\n\t\tBern       ---  4\tNaples  --- 13" );
        printf( "\n\t\tBrussels   ---  5\tParis   --- 14" );
        printf( "\n\t\tBudapest   ---  6\tPrague  --- 15" );
        printf( "\n\t\tCopenhagen ---  7\tRome    --- 16" );
        printf( "\n\t\tGenoa      ---  8\tTrieste --- 17" );
        printf( "\n\t\tHamburg    ---  9\tWarsaw  --- 18" );
        printf( "\n\n\t\t" );
        scanf( "%d", &START );

        /* Adjust index for any city except Warsaw, whose
           index is 18. */
        if ( START < 18 )
           --START;
}

void failure( void )
{
        printf( "\n\n\n\tYou can't get there "
                "from here.\n\n " );
}

void success( const CITY* ptr )
{
        int i;

        printf( "\n\n\n" );
        printf( "\n\n\tOPTIMAL PATH LENGTH: %d\n",
                ptr -> g_val );
        printf( "\n\n\tOPTIMAL PATH:\n" );
        print_cities( ptr );

        printf( "\n\n\tEXPANDED VERTICES:\n" );
```

```
        for ( i = 0; i < SIZE; ++i )
            if ( cities[ i ].closed == TRUE )
               printf( "\n\t%s", cities[ i ].name );
        printf( "\n\n\n" );
}

void print_cities( const CITY* ptr )
{
        if ( ptr -> ind == START ) /* halt backtrack */
           printf( "\n\t%s", ptr -> name );
        else {
           print_cities( ptr -> backptr ); /* backtrack */
           printf( "\n\t%s", ptr -> name );
        }
}

void set_adjacency_lists( void )
{
   /***         Amsterdam to ...     ***/
   cities[ 0 ].adjcs[ 0 ] = &cities[ 3 ];    /* Bern */
   cities[ 0 ].adjcs[ 1 ] = &cities[ 4 ];    /* Brussels */
   cities[ 0 ].adjcs[ 2 ] = &cities[ 8 ];    /* Hamburg */
   cities[ 0 ].adjcs[ 3 ] = &cities[ 11 ];   /* Munich */

   /***         Belgrade to ...      ***/
   cities[ 1 ].adjcs[ 0 ] = &cities[ 5 ];    /* Budapest */
   cities[ 1 ].adjcs[ 1 ] = &cities[ 16 ];   /* Trieste */
   cities[ 1 ].adjcs[ 2 ] = &cities[ 17 ];   /* Vienna */

   /***         Berlin to ...        ***/
   cities[ 2 ].adjcs[ 0 ] = &cities[ 3 ];    /* Bern */
   cities[ 2 ].adjcs[ 1 ] = &cities[ 8 ];    /* Hamburg */
   cities[ 2 ].adjcs[ 2 ] = &cities[ 14 ];   /* Prague */
   cities[ 2 ].adjcs[ 3 ] = &cities[ 18 ];   /* Warsaw */
```

```
/***          Bern to ...          ***/
cities[ 3 ].adjcs[ 0 ] = &cities[ 0 ];    /* Amsterdam */
cities[ 3 ].adjcs[ 1 ] = &cities[ 2 ];    /* Belgrade */
cities[ 3 ].adjcs[ 2 ] = &cities[ 4 ];    /* Brussels */
cities[ 3 ].adjcs[ 3 ] = &cities[ 7 ];    /* Genoa */
cities[ 3 ].adjcs[ 4 ] = &cities[ 10 ];   /* Madrid */
cities[ 3 ].adjcs[ 5 ] = &cities[ 11 ];   /* Munich */
cities[ 3 ].adjcs[ 6 ] = &cities[ 16 ];   /* Trieste */

/***          Brussels to ...      ***/
cities[ 4 ].adjcs[ 0 ] = &cities[ 0 ];    /* Amsterdam */
cities[ 4 ].adjcs[ 1 ] = &cities[ 3 ];    /* Bern */
cities[ 4 ].adjcs[ 2 ] = &cities[ 7 ];    /* Genoa */
cities[ 4 ].adjcs[ 3 ] = &cities[ 13 ];   /* Paris */

/***          Budapest to ...      ***/
cities[ 5 ].adjcs[ 0 ] = &cities[ 1 ];    /* Belgrade */
cities[ 5 ].adjcs[ 1 ] = &cities[ 16 ];   /* Trieste */
cities[ 5 ].adjcs[ 2 ] = &cities[ 17 ];   /* Vienna */
cities[ 5 ].adjcs[ 3 ] = &cities[ 18 ];   /* Warsaw */

/***          Copenhagen to ...    ***/
cities[ 6 ].adjcs[ 0 ] = &cities[ 8 ];    /* Hamburg */

/***          Genoa to ...         ***/
cities[ 7 ].adjcs[ 0 ] = &cities[ 4 ];    /* Brussels */
cities[ 7 ].adjcs[ 1 ] = &cities[ 10 ];   /* Madrid */
cities[ 7 ].adjcs[ 2 ] = &cities[ 13 ];   /* Paris */
cities[ 7 ].adjcs[ 3 ] = &cities[ 15 ];   /* Rome */
cities[ 7 ].adjcs[ 4 ] = &cities[ 16 ];   /* Trieste */

/***          Hamburg to ...       ***/
cities[ 8 ].adjcs[ 0 ] = &cities[ 0 ];    /* Amsterdam */
cities[ 8 ].adjcs[ 1 ] = &cities[ 2 ];    /* Berlin */
cities[ 8 ].adjcs[ 2 ] = &cities[ 6 ];    /* Copenhagen */
```

```
/***          Lisbon to ...         ***/
cities[ 9 ].adjcs[ 0 ] = &cities[ 10 ];   /* Madrid */

/***           Madrid to ...        ***/
cities[ 10 ].adjcs[ 0 ] = &cities[ 3 ];   /* Bern */
cities[ 10 ].adjcs[ 1 ] = &cities[ 7 ];   /* Genoa */
cities[ 10 ].adjcs[ 2 ] = &cities[ 9 ];   /* Lisbon */
cities[ 10 ].adjcs[ 3 ] = &cities[ 13 ];  /* Paris */

/***          Munich to ...         ***/
cities[ 11 ].adjcs[ 0 ] = &cities[ 0 ];   /* Amsterdam */
cities[ 11 ].adjcs[ 1 ] = &cities[ 14 ];  /* Prague */
cities[ 11 ].adjcs[ 2 ] = &cities[ 15 ];  /* Rome */
cities[ 11 ].adjcs[ 3 ] = &cities[ 17 ];  /* Vienna */

/***           Naples to ...         ***/
cities[ 12 ].adjcs[ 0 ] = &cities[ 15 ];  /* Rome */

/***           Paris to ...         ***/
cities[ 13 ].adjcs[ 0 ] = &cities[ 4 ];   /* Brussels */
cities[ 13 ].adjcs[ 1 ] = &cities[ 7 ];   /* Genoa */
cities[ 13 ].adjcs[ 2 ] = &cities[ 10 ];  /* Madrid */

/***          Prague to ...         ***/
cities[ 14 ].adjcs[ 0 ] = &cities[ 2 ];   /* Berlin */
cities[ 14 ].adjcs[ 1 ] = &cities[ 11 ];  /* Munich */
cities[ 14 ].adjcs[ 2 ] = &cities[ 17 ];  /* Vienna */
cities[ 14 ].adjcs[ 3 ] = &cities[ 18 ];  /* Warsaw */

/***           Rome to ...          ***/
cities[ 15 ].adjcs[ 0 ] = &cities[ 7 ];   /* Genoa */
cities[ 15 ].adjcs[ 1 ] = &cities[ 11 ];  /* Munich */
cities[ 15 ].adjcs[ 2 ] = &cities[ 12 ];  /* Naples */
cities[ 15 ].adjcs[ 3 ] = &cities[ 16 ];  /* Trieste */
```

```
    /***           Trieste to ...       ***/
    cities[ 16 ].adjcs[ 0 ] = &cities[ 1 ];   /* Belgrade */
    cities[ 16 ].adjcs[ 1 ] = &cities[ 3 ];   /* Bern */
    cities[ 16 ].adjcs[ 2 ] = &cities[ 5 ];   /* Budapest */
    cities[ 16 ].adjcs[ 3 ] = &cities[ 7 ];   /* Genoa */
    cities[ 16 ].adjcs[ 4 ] = &cities[ 15 ];  /* Rome */
    cities[ 16 ].adjcs[ 5 ] = &cities[ 17 ];  /* Vienna */

    /***           Warsaw to ...        ***/
    cities[ 18 ].adjcs[ 0 ] = &cities[ 2 ];   /* Berlin */
    cities[ 18 ].adjcs[ 1 ] = &cities[ 5 ];   /* Budapest */
    cities[ 18 ].adjcs[ 2 ] = &cities[ 14 ];  /* Prague */
    cities[ 18 ].adjcs[ 3 ] = &cities[ 17 ];  /* Vienna */
}
```

Section 11.4: Fractals

```
/* This program uses Borland graphics to
   draw the von Koch snowflake at a user-
   specified level.
*/

#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <ctype.h>

/* factor is set to sqrt( 3 ) / 2 by
   graph_vonkoch and is used by
   vonkoch
*/
double factor;

char inbuff[ 80 ]; /* for user input */

void vonkoch( int level, int scale, int rot );
void graph_vonkoch( void );

main()
{
   for ( ; ; ) {
      printf( "\nDo another (y = yes, n = no)? " );
      gets( inbuff );
      if ( toupper( *inbuff ) != 'Y' )
         return EXIT_SUCCESS;
      graph_vonkoch();
   }
}
```

```
/* graph_vonkoch sets up the graphics,
   obtains the level from the user,
   and invokes vonkoch to draw the snowflake.
*/
void graph_vonkoch( void )
{
   int gdriver = DETECT, gmode, errorcode, level;

   printf( "Enter level: " );
   gets( inbuff );
   level = atoi( inbuff );

   /* gdriver is set to DETECT, which asks
      the system to autodetect the proper
      graphics driver. gmode will be set
      to the highest resolution of the
      driver selected. "" means use
      current directory to look for the
      graphics driver. */
   initgraph( &gdriver, &gmode, "" );

   errorcode = graphresult();
   if ( errorcode != grOk )
   {
      printf( "Graphics error: %s\n",
              grapherrormsg( errorcode ) );
      return;
   }

   factor = sqrt( 3 ) / 2;
   moveto( 0, 0 );
   /* Draw snowflake.
      level = user-specified level
      getmaxx() = horizontal resolution
```

```
      0 = start drawing to the right */
   vonkoch( level, getmaxx(), 0 );

   /* getmaxy() = vertical resolution,
      so print message halfway down. */
   outtextxy( 0, getmaxy() / 2, "Press any key to continue" );
   getch(); /* get char, no echo */
   closegraph();
}

/* Draw snowflake at the specified level, to the specified
   scale, rot*60 degrees clockwise from the horizontal.
   level == 0, which is a straight line, is the most
   primitive level. If level > 0, each line segment in

                            _  _
                             \/

   is replaced by a von Koch snowflake of level n - 1.

   linerel(x,y) (draw line relative) draws a line from
   the current pixel position (p,q) to (p + x, q + y).
*/
void vonkoch( int level, int scale, int rot )
{
   if ( level == 0 )
      switch ( rot ) {
      case 0:
         linerel( scale, 0 );
         return;
      case 1:
         linerel( scale / 2, scale * factor );
         return;
      case 2:
         linerel( -scale / 2, scale * factor );
         return;
```

```
      case 3:
         linerel( -scale, 0 );
         return;
      case 4:
         linerel( -scale / 2, -scale * factor );
         return;
      case 5:
         linerel( scale / 2, -scale * factor );
         return;
      }

   vonkoch( level - 1, scale / 3, rot );
   vonkoch( level - 1, scale / 3, ( rot + 1 ) % 6 );
   vonkoch( level - 1, scale / 3, ( rot + 5 ) % 6 );
   vonkoch( level - 1, scale / 3, rot );
}
```