```c
/*
 * Задание 1.5.  Даны координаты вершин некоторого треугольника и координаты
 * некоторой точки внутри него.  Найти расстояние от этой точки до ближайшей
 * стороны треугольника.
 */

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

typedef struct _Point {
  double x;
  double y;
} point, *pointptr;


point Pt, VtPt1, VtPt2, VtPt3;

double GetDistance(pointptr Pt0, pointptr Pt1, pointptr Pt2);


int main()
{
  double dist, min1, min2;

  fputs("Enter point x coordinate: ", stdout);
  scanf("%lf", &Pt.x);
  fputs("Enter point y coordinate: ", stdout);
  scanf("%lf", &Pt.y);

  fputs("\nEnter vertex 1 x coordinate: ", stdout);
  scanf("%lf", &VtPt1.x);
  fputs("Enter vertex 1 y coordinate: ", stdout);
  scanf("%lf", &VtPt1.y);

  fputs("Enter vertex 2 x coordinate: ", stdout);
  scanf("%lf", &VtPt2.x);
  fputs("Enter vertex 2 y coordinate: ", stdout);
  scanf("%lf", &VtPt2.y);

  fputs("Enter vertex 3 x coordinate: ", stdout);
  scanf("%lf", &VtPt3.x);
  fputs("Enter vertex 3 y coordinate: ", stdout);
  scanf("%lf", &VtPt3.y);

  dist = GetDistance(&Pt, &VtPt1, &VtPt2);
  min1 = GetDistance(&Pt, &VtPt1, &VtPt3);
  min2 = GetDistance(&Pt, &VtPt2, &VtPt3);
  if ( min1 < dist )
    dist = min1;
  if ( min2 < dist )
    dist = min2;

  printf("\nDistance from point (%.3lf,%.3lf) to the closest side of the triangle: %.3lf\n",
         Pt.x, Pt.y, dist);

  return 0;
}
```

```
/* Calculates distance between a point Pt0 and a line (designated by two
 * points Pt1 and Pt2 it passes through) */
double GetDistance(pointptr Pt0, pointptr Pt1, pointptr Pt2)
{
  double X2MinusX1, Y2MinusY1;

  X2MinusX1 = Pt2->x - Pt1->x;
  Y2MinusY1 = Pt2->y - Pt1->y;

  return
    fabs( ( Pt0->x - Pt1->x ) * Y2MinusY1 - ( Pt0->y - Pt1->y ) * X2MinusX1 ) /
    sqrt(Y2MinusY1*Y2MinusY1 + X2MinusX1*X2MinusX1);
}



/*
 * Local variables:
 * compile-command: "gcc -lm 1_05.c && ./a.out <<< \"1.6 1 1 0 1 3 3 0\""
 * End:
 */
```

```c
/*
 * Задание 1.16.  Составить процедуру построения строки символов, являющейся
 * записью заданного числа в десятичной системе счисления; строка должна
 * содержать заданное число символов после запятой.
 */

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

#define DECIMAL_POINT   '.'
#define MEM_CHUNK_SIZE  16

char *ftoa(double num, unsigned DecimalPlaces);

int main()
{
  double num;
  int dp;
  char *str;

  fputs("Enter a FP number: ", stdout);
  scanf("%lf", &num);
  fputs("Enter number of decimal places: ", stdout);
  scanf("%u", &dp);

  if ( !(str = ftoa(num, dp)) ) {
    puts("Error converting the number.");
    return -1;
  }

  puts(str);

  return 0;
}



int CheckBufferSize(char **str, int *len, int cur);
char* strrev(char *str, unsigned n);

char *ftoa(double num, unsigned DecimalPlaces)
{
  char *str;
  int len, i, sgn;
  double IntPart, FracPart;

  /* Allocate initial memory chunk */
  if ( !(str = (char *) malloc(MEM_CHUNK_SIZE)) )
    return NULL;
  len = MEM_CHUNK_SIZE;
  i = 0;
  /* Save sign and store absolute value in num */
  if ( num < 0 ) {
    sgn = 1;
    num = -num;
  } else
    sgn = 0;
  /* Split up num in integer and fractional parts */
```

```
  FracPart = modf(num, &IntPart);

  /* Get integer part (in reversed order) */
  while ( IntPart >= 1 ) {
    *(str + i++) = '0' + (char) fmod(IntPart, 10);
    IntPart = IntPart / 10;
    if ( !CheckBufferSize(&str, &len, i) )
      return NULL;
  }
  /* Add sign if necessary */
  if (sgn) {
    *(str + i++) = '-';
    if ( !CheckBufferSize(&str, &len, i) )
      return NULL;
  }
  strrev(str, i);

  /* Add decimal point if necessary */
  if ( DecimalPlaces ) {
    *(str + i++) = DECIMAL_POINT;
    if ( !CheckBufferSize(&str, &len, i) )
      return NULL;
  }
  /* Get fractional part */
  while ( DecimalPlaces-- ) {
    IntPart = FracPart * 10;
    *(str + i++) = '0' + (char) IntPart;
    FracPart = modf(IntPart, &IntPart);
    if ( !CheckBufferSize(&str, &len, i) )
      return NULL;
  }

  /* Terminate the string */
  *(str + i) = '\0';

  return str;
}

/* Grows buffer if necessary.  If reallocation fails, frees the
 * original buffer.  Returns buffer length on success, zero on
 * failure. */
int CheckBufferSize(char **str, int *len, int cur)
{
  char *tmp;

  if ( cur >= *len ) {
    *len += MEM_CHUNK_SIZE;
    if ( tmp = (char *) realloc(*str, *len ) ) {
      *str = tmp;
      return *len;
    }
    else {
      free(*str);
      return 0;
    }
  }

  return *len;
}
```

```
char *strrev(char *str, unsigned n)
{
  char *rev, c;

  if ( !str )
    return NULL;

  rev = str + n - 1;
  while ( rev > str ) {
    c = *rev;
    *rev-- = *str;
    *str++ = c;
  }

  return str;
}



/*
 * Local variables:
 * compile-command: "gcc -lm 1_16.c && ./a.out <<< '5.84 3'"
 * End:
 */
```

```
/*
 * Задание 2.1.  Дано выражение a*b+c*d.  Организовать вычисление этого
 * выражения, используя алгоритм польской записи.  Применить программный стек.
 */

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

#define LINE_BUFFER_LENGTH  256
#define STACK_MEM_CHUNK     16

const char *strGreeting="Reverse polish integer calculator.\n\
Enter expression(s) to calculate (Ctrl-D to quit):\n";

const char *strStackCleared="stack cleared\n";
const char *strErrMem="error: not enough memory\n";
const char *strErrStackEmpty="error: stack empty\n";
const char *strErrUnknownOperation="error: unknown operation `%c'\n";
const char *strErrUnknownCommand="error: unknown command `%c'\n";

char *line;

void ParseLine(char *line);
int  ParseCommand(char **ptr);
int  PopOneOp(long *ptrOp1);
int  PopTwoOps(long *ptrOp1, long *ptrOp2);
int  PushAndPrintResult(long num);

int  Push(long Value);
int  Pop(long *Value);
int  GetTop(long *Value);
void ClearStack(void);
void PrintStack(void);
void FreeStack(void);



int main()
{
  fputs(strGreeting, stdout);

  if ( !(line = malloc(LINE_BUFFER_LENGTH)) ) {
    fputs(strErrMem, stderr);
    return -1;
  }

  while ( fgets(line, LINE_BUFFER_LENGTH, stdin) ) {
    ParseLine(line);
  }

  FreeStack();

  return 0;
}



void ParseLine(char *line)
{
```

```
  char *ptr;
  long op1, op2;
  int  sgn, BailOut;

  ptr = line;

  BailOut = 0;
  while ( *ptr && !BailOut ) {
    if ( isspace(*ptr) ) {
      /* Skip spaces */
      ptr++;

    } else if ( isalpha(*ptr) ) {
      /* Parse out the command */
      if ( !ParseCommand(&ptr) )
        BailOut = -1;

    } else if ( isdigit(*ptr) || *ptr == '_' ) {
      /* Parse out the number */
      if ( *ptr == '_' ) {
        sgn = -1;
        ptr++;
      } else
        sgn = 0;
      op1 = strtol(ptr, &ptr, 0);
      if ( sgn )
        op1 = -op1;
      if ( !Push(op1) ) {
        fputs(strErrMem, stderr);
        BailOut = -1;
      }

    } else {
      /* Parse out the operation */
      switch ( *ptr ) {
      case '+':
        (BailOut = PopTwoOps(&op1, &op2)) ||
          (BailOut = PushAndPrintResult(op1+op2));
        break;
      case '-':
        (BailOut = PopTwoOps(&op1, &op2)) ||
          (BailOut = PushAndPrintResult(op1-op2));
        break;
      case '*':
        (BailOut = PopTwoOps(&op1, &op2)) ||
          (BailOut = PushAndPrintResult(op1*op2));
        break;
      case '/':
        (BailOut = PopTwoOps(&op1, &op2)) ||
          (BailOut = PushAndPrintResult(op1/op2));
        break;
      case '\%': /* modulo */
        (BailOut = PopTwoOps(&op1, &op2)) ||
          (BailOut = PushAndPrintResult(op1%op2));
        break;
      case '$': /* absolute value */
        (BailOut = PopOneOp(&op1)) ||
          ( BailOut = PushAndPrintResult( (op1 < 0) ? -op1 : op1 ) );
        break;
```

```
      case '~': /* negate */
        (BailOut = PopOneOp(&op1)) || ( BailOut = PushAndPrintResult(-op1) );
        break;
      default:
        fprintf(stderr, strErrUnknownOperation, *ptr);
        BailOut = -1;
        break;
      }
      ptr++;
    }
  }

  return;
}

/* Returns -1 on success, 0 if command is unknown */
int ParseCommand(char **ptr)
{
  long num, op1;
  int ret;

  ret = -1;
  switch (**ptr) {
  case 't': /* Print top stack value */
    if ( !GetTop(&num) )
      fprintf(stderr, strErrStackEmpty);
    else
      printf("%ld\n", num);
    break;
  case 's': /* Print entire stack */
    PrintStack();
    break;
  case 'p': /* Pop off top stack value */
    if ( !Pop(&num) )
      fprintf(stderr, strErrStackEmpty);
    else
      printf("%ld\n", num);
    break;
  case 'c': /* Clear entire stack */
    ClearStack();
    fputs(strStackCleared, stdout);
    break;
  case 'e': /* Exchange two top values in the stack */
    PopTwoOps(&num, &op1) || PushAndPrintResult(op1) || PushAndPrintResult(num);
    break;
  case 'd': /* Duplicate top value */
    if ( !GetTop(&num) )
      fprintf(stderr, strErrStackEmpty);
    else
      PushAndPrintResult(num);
    break;
  default:
    fprintf(stderr, strErrUnknownCommand, **ptr);
    ret = 0;
    break;
  }

  (*ptr)++;
```

```
  return ret;
}

int PopOneOp(long *ptrOp1)
{
  if ( !Pop(ptrOp1) ) {
    fputs(strErrStackEmpty, stderr);
    return -1;
  }

  return 0;
}

int PopTwoOps(long *ptrOp1, long *ptrOp2)
{
  if ( !Pop(ptrOp2) || !Pop(ptrOp1) ) {
    fputs(strErrStackEmpty, stderr);
    return -1;
  }

  return 0;
}

int PushAndPrintResult(long num)
{
  if ( !Push(num) ) {
    fputs(strErrMem, stderr);
    return -1;
  }

  printf("%ld\n", num);
  return 0;
}



static long *Stack = NULL;
static int StackTop = -1;
static int StackLen = 0;

/* Returns length of stack on success, 0 on failure to grow stack */
int Push(long Value)
{
  long *ptr;

  /* Grow stack if necessary */
  if ( ++StackTop >= StackLen ) {
    StackLen += STACK_MEM_CHUNK;
    if ( !(ptr = (long *) realloc(Stack, StackLen * sizeof(*Stack))) ) {
      StackLen -= STACK_MEM_CHUNK;
      return 0;
    }
    Stack = ptr;
  }
  /* Push the value onto stack */
  *(Stack + StackTop) = Value;

  return StackTop + 1;
}
```

```
/* Returns -1 on success, 0 if stack is empty */
int Pop(long *Value)
{
  if ( StackTop >= 0 ) {
    *Value = *(Stack + StackTop--);
    return -1;
  }

  return 0;
}

/* Returns -1 on success, 0 if stack is empty */
int GetTop(long *Value)
{
  if ( StackTop >= 0 ) {
    *Value = *(Stack + StackTop);
    return -1;
  }

  return 0;
}

void ClearStack(void)
{
  StackTop = -1;
}

void PrintStack(void)
{
  int i;

  if (StackTop < 0) {
    fputs(strErrStackEmpty, stderr);
    return;
  }

  i = 0;
  while ( i <= StackTop )
    printf("%ld\n", *(Stack + i++));
}

void FreeStack(void)
{
  free(Stack);
  Stack = NULL;
  StackLen = 0;
  StackTop = -1;
}



/*
 * Local variables:
 * compile-command: "gcc 2_01.c"
 * End:
 */
```

```
/*
 * Задание 2.5.  Произвести проверку соблюдения баланса скобок вида '[', ']' в
 * арифметическом выражении.  Использовать программный стек.
 */

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

#define STACK_MEM_CHUNK     16


const char *strClosingParen=")}]";
const char *strOpeningParen="({[";

const char *strErrOpenFile           ="error: cannont open file '%s'\n";
const char *strErrMissingClosing     ="error: missing '%c'\n";
const char *strErrTooManyClosing     ="L%ld-C%ld: too many '%c'\n";
const char *strErrMismatchingClosing="L%ld-C%ld: mismatching '%c'\n";

int  PopAndCheckParen(int ClosingID, unsigned long line, unsigned long col);

int  Push(int Value);
int  Pop(int *Value);
void FreeStack(void);


int main(int argc, char **argv)
{
  int c, id, ret;
  unsigned long line, col;

  /* If user provided filename, try to redirect stdin to that file */
  if ( argc > 1 ) {
    if ( !freopen(*(argv+1), "r", stdin) ) {
      fprintf(stderr, strErrOpenFile, *(argv+1));
      return -1;
    }
  }

  /* Main loop */
  ret = 0;
  line = 1;
  col = 0;
  while ( !ret && (c=getchar()) != EOF ) {

    /* Keep track of line and column */
    if ( c == '\n' ) {
      line++;
      col = 0;
    } else
      col++;

    switch ( c ) {
    case '(':
      Push(0);
      break;
    case ')':
      ret = PopAndCheckParen(0, line, col);
      break;
```

```
    case '{':
      Push(1);
      break;
    case '}':
      ret = PopAndCheckParen(1, line, col);
      break;
    case '[':
      Push(2);
      break;
    case ']':
      ret = PopAndCheckParen(2, line, col);
      break;
    default:
      break;
    }
  }

  if ( !ret ) {
    if ( Pop(&id) ) {
      fprintf(stderr, strErrMissingClosing, *(strClosingParen + id));
      ret = -1;
    } else
      fputs("all parentheses matched\n", stdout);
  }

  FreeStack();

  return ret;
}

int PopAndCheckParen(int ClosingID, unsigned long line, unsigned long col)
{
  int PoppedID;

  if ( !Pop(&PoppedID) ) {
    fprintf(stderr, strErrTooManyClosing,
            line, col, *(strClosingParen + ClosingID));
    return -1;
  }
  if ( PoppedID != ClosingID ) {
    fprintf(stderr, strErrMismatchingClosing,
            line, col, *(strClosingParen+ClosingID));
    return -1;
  }

  return 0;
}



static int *Stack;
static int StackTop = -1;
static int StackLen = 0;

/* Returns length of stack on success, 0 on failure to grow stack */
int Push(int Value)
{
  int *ptr;
```

```
  /* Grow stack if necessary */
  if ( ++StackTop >= StackLen ) {
    StackLen += STACK_MEM_CHUNK;
    if ( !(ptr = (int *) realloc(Stack, StackLen * sizeof(*Stack))) ) {
      StackLen -= STACK_MEM_CHUNK;
      return 0;
    }
    Stack = ptr;
  }
  /* Push the value onto stack */
  *(Stack + StackTop) = Value;

  return StackTop + 1;
}

/* Returns -1 on success, 0 if stack is empty */
int Pop(int *Value)
{
  if ( StackTop >= 0 ) {
    *Value = *(Stack + StackTop--);
    return -1;
  }

  return 0;
}

void FreeStack(void)
{
  free(Stack);
  Stack = NULL;
  StackLen = 0;
  StackTop = -1;
}



/*
 * Local variables:
 * compile-command: "gcc 2_05.c && ./a.out 2_05.in"
 * End:
 */
```

```
/*
 * Задание 3.5.  Для ряда натуральных чисел длиной N>2, представленного в виде
 * списка, построить последовательность:
 *
 *    A_1 * A_N, A_2 * A_(N-1), ..., A_[(N+1)/2] * A_[N/2+1]
 */

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

typedef struct _item {
  int ItemKey;
  int ItemValue;
  struct _item *ptrNext;
} item, *itemptr;

itemptr MakeUpList(int n);
int    *CalculateResult(itemptr ptrHead, int n);
void    PrintOut(itemptr ptrHead, int *ptrResult, int n);
void    FreeListMem(itemptr ptrHead);



int main()
{
  int n;
  itemptr ptrHead;
  int *ptrResult;

  /* Seed the random number generator */
  srand(time(0));

  /* Get list length */
  fputs("Enter list length: ", stdout);
  scanf("%d", &n);

  /* Make up a list with arbitrary contents */
  if ( !(ptrHead = MakeUpList(n)) ) {
    puts("Error allocating memory for the list.");
    return -1;
  }

  /* Allocate memory for the result buffer */
  if ( !(ptrResult = CalculateResult(ptrHead, n)) ) {
    puts("Error allocating memory for the result.");
    FreeListMem(ptrHead);
    return -1;
  }

  /* Print out the list and the result */
  PrintOut(ptrHead, ptrResult, n);

  /* Free the memory taken up by the list and result buffer */
  FreeListMem(ptrHead);
  free(ptrResult);

  /* Bail out */
  return 0;
}
```

```
itemptr MakeUpList(int n)
{
  int i;
  itemptr ptrHead, ptrPrev, ptrCurr;

  ptrHead = (itemptr) malloc(sizeof(item));
  if ( !ptrHead )
    return NULL;
  ptrHead->ItemKey = 0;
  ptrHead->ItemValue = rand() / (RAND_MAX/10);
  ptrPrev = ptrHead;

  for ( i = 1; i < n; i++ ) {
    ptrCurr = (itemptr) malloc(sizeof(item));
    if ( !ptrCurr ) {
      ptrPrev->ptrNext = NULL;
      FreeListMem(ptrHead);
      return NULL;
    }
    ptrCurr->ItemKey = i;
    /* Make sure the values are sane */
    ptrCurr->ItemValue = rand() / (RAND_MAX/10);
    ptrPrev->ptrNext = ptrCurr;
    ptrPrev = ptrCurr;
  }
  ptrPrev->ptrNext = NULL;

  return ptrHead;
}



int *CalculateResult(itemptr ptrHead, int n)
{
  int i, tmp;
  itemptr ptrCurr;
  int *ptrResult;

  ptrResult = (int *) malloc( sizeof(int) * ((n+1)/2) );
  if ( !ptrResult )
    return NULL;

  ptrCurr = ptrHead;
  /* Go through the first half of the list putting items into result
     buffer */
  tmp = (n+1) / 2;
  for ( i = 0; i < tmp; i++ ) {
    *(ptrResult + i) = ptrCurr->ItemValue;
    ptrCurr = ptrCurr->ptrNext;
  }
  i--;
  /* Continue going through the list, now multiplying items stored in
   * result buffer (in backwards order) by items in the second half of
   * the list. */
  if ( n & 0x01 ) {
    /* If number of items is odd, first square up the central item */
```

```
    *(ptrResult + i) *= *(ptrResult + i);
    i--;
  }
  for ( ; i >= 0; i-- ) {
    *(ptrResult + i) *= ptrCurr->ItemValue;
    ptrCurr = ptrCurr->ptrNext;
  }

  return ptrResult;
}



void PrintOut(itemptr ptrHead, int *ptrResult, int n)
{
  int i;

  puts("Original list:");
  while ( ptrHead ) {
    printf("%2d: %3d\n", ptrHead->ItemKey, ptrHead->ItemValue);
    ptrHead = ptrHead->ptrNext;
  }
  puts("Resulting list:");
  for ( i = 0, n = (n+1) / 2; i < n; i++)
    printf("%2d:%5d\n", i, *(ptrResult + i));
}



void FreeListMem(itemptr ptrHead)
{
  itemptr ptrTemp;

  while ( ptrHead ) {
    ptrTemp = ptrHead->ptrNext;
    free((void *) ptrHead);
    ptrHead = ptrTemp;
  }
}



/*
 * Local variables:
 * compile-command: "gcc 3_05.c && ./a.out <<< 7"
 * End:
 */
```

```
/*
 * Задание 3.16.  Описать процедуру, которая проверяет, входит ли список L1 в
 * список L2.
 */

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

typedef struct _item {
  int ItemKey;
  int ItemValue;
  struct _item *ptrNext;
} item, *itemptr;

itemptr MakeUpList(int n, int *ptrItems);
itemptr lstlst(itemptr ptrHaystack, itemptr ptrNeedle);
void    PrintOut(itemptr ptrHead);
void    FreeListMem(itemptr ptrHead);


int ptrItems1[] = {  2, 5, 2, 7, 1, 8, 6, 4, 8, 9, 7 };
int ptrItems2[] = {           7, 1, 8                 };

const char *strErrListMem = "Error allocating memory for a list.";
const char *strPrintFormat = "\nSecond list is %sa sublist of the first.\n";



int main()
{
  itemptr ptrHead1, ptrHead2;

  /* Make up two lists on the basis of the arrays */
  if ( !(ptrHead1 =
         MakeUpList(sizeof(ptrItems1)/sizeof(ptrItems1[0]), ptrItems1)) ) {
    puts(strErrListMem);
    return -1;
  }
  if ( !(ptrHead2 =
         MakeUpList(sizeof(ptrItems2)/sizeof(ptrItems2[0]), ptrItems2)) ) {
    puts(strErrListMem);
    FreeListMem(ptrHead1);
    return -1;
  }

  /* Print out the lists */
  puts("\n List 1:");
  PrintOut(ptrHead1);
  puts("\n List 2:");
  PrintOut(ptrHead2);

  /* See if the second list is a sublist of the first */
  printf(strPrintFormat, lstlst(ptrHead1, ptrHead2) ? "": "not ");

  /* Free the memory taken up by the lists */
  FreeListMem(ptrHead1);
  FreeListMem(ptrHead2);

  /* Bail out */
```

```
  return 0;
}



itemptr lstlst(itemptr ptrHaystack, itemptr ptrNeedle)
{
  int iNeedleFirstItem;
  itemptr ptrNeedleCurr, ptrHaystackCurr, ptrSublist;

  if ( !ptrNeedle )
    return NULL;

  /* To avoid dereferencing pointer to needle head in each loop, we
   * store first needle item in a local variable */
  ptrSublist = NULL;
  iNeedleFirstItem = ptrNeedle->ItemValue;
  while ( ptrHaystack ) {
    if ( ptrHaystack->ItemValue == iNeedleFirstItem ) {
      ptrNeedleCurr = ptrNeedle->ptrNext;
      ptrHaystackCurr = ptrHaystack->ptrNext;
      while ( ptrNeedleCurr &&
              ptrHaystackCurr &&
              ptrNeedleCurr->ItemValue == ptrHaystackCurr->ItemValue ) {
        ptrHaystackCurr = ptrHaystackCurr->ptrNext;
        ptrNeedleCurr = ptrNeedleCurr->ptrNext;
      }
      if ( !ptrNeedleCurr ) {
        ptrSublist = ptrHaystack;
        break;
      }
    }
    ptrHaystack = ptrHaystack->ptrNext;
  }

  return ptrSublist;
}



itemptr MakeUpList(int n, int *ptrItems)
{
  int i;
  itemptr ptrHead, ptrPrev, ptrCurr;

  ptrHead = (itemptr) malloc(sizeof(item));
  if ( !ptrHead )
    return NULL;
  ptrHead->ItemKey = 0;
  ptrHead->ItemValue = *ptrItems;
  ptrPrev = ptrHead;

  for ( i = 1; i < n; i++ ) {
    ptrCurr = (itemptr) malloc(sizeof(item));
    if ( !ptrCurr ) {
      ptrPrev->ptrNext = NULL;
      FreeListMem(ptrHead);
      return NULL;
    }
```

```
    ptrCurr->ItemKey = i;
    ptrCurr->ItemValue = *(ptrItems + i);
    ptrPrev->ptrNext = ptrCurr;
    ptrPrev = ptrCurr;
  }
  ptrPrev->ptrNext = NULL;

  return ptrHead;
}



void PrintOut(itemptr ptrHead)
{
  while ( ptrHead ) {
    printf("%2d:%5d\n", ptrHead->ItemKey, ptrHead->ItemValue);
    ptrHead = ptrHead->ptrNext;
  }
}



void FreeListMem(itemptr ptrHead)
{
  itemptr ptrTemp;

  while ( ptrHead ) {
    ptrTemp = ptrHead->ptrNext;
    free((void *) ptrHead);
    ptrHead = ptrTemp;
  }
}



/*
 * Local variables:
 * compile-command: "gcc 3_16.c && ./a.out"
 * End:
 */
```

```
/*
 * Задание 4.5.  Используя двунаправленный список, содержащий символы и их
 * шифры, зашифровать текст, содержащийся в файле.
 */

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

typedef struct _litem {
  char Key;
  char Value;
  struct _litem *ptrNext;
  struct _litem *ptrPrev;
} litem, *litemptr;

litemptr MakeUpList();
litemptr lstlst(litemptr ptrHead, litemptr ptrCurr, char c);
void     FreeListMem(litemptr ptrHead);


const char *strErrListMem ="error: cannot allocate memory for the list\n";
const char *strErrOpenFile="error: cannot open file `%s'\n";


int main(int argc, char **argv)
{
  litemptr ptrHead, ptrCurr;
  char c;

  /* If user provided filename, try to redirect stdin to that file */
  if ( argc > 1 ) {
    if ( !freopen(*(argv+1), "r", stdin) ) {
      fprintf(stderr, strErrOpenFile, *(argv+1));
      return -1;
    }
  }

  /* Make up the list with character codes */
  if ( !(ptrHead = MakeUpList()) ) {
    fputs(strErrListMem, stderr);
    return -1;
  }

  ptrCurr = ptrHead;
  while ( (c=getchar()) != EOF ) {
    ptrCurr = lstlst(ptrHead, ptrCurr, c);
    putchar(ptrCurr->Value);
  }

  /* Free the list memory */
  FreeListMem(ptrHead);

  /* Bail out */
  return 0;
}


litemptr MakeUpList()
{
```

```
  int i;
  litemptr ptrHead, ptrPrev, ptrCurr;

  ptrHead = (litemptr) malloc(sizeof(litem));
  if ( !ptrHead )
    return NULL;
  ptrHead->Key   = 0;
  ptrHead->Value = 1;
  ptrHead->ptrPrev   = NULL;
  ptrPrev = ptrHead;

  for ( i = 1; i < 256; i++ ) {
    ptrCurr = (litemptr) malloc(sizeof(litem));
    if ( !ptrCurr ) {
      ptrPrev->ptrNext = NULL;
      FreeListMem(ptrHead);
      return NULL;
    }
    ptrCurr->Key = i;
    ptrCurr->Value = (i + 1) % 256;
    ptrCurr->ptrPrev = ptrPrev;
    ptrPrev->ptrNext = ptrCurr;
    ptrPrev = ptrCurr;
  }
  ptrPrev->ptrNext = NULL;

  return ptrHead;
}


litemptr lstlst(litemptr ptrHead, litemptr ptrCurr, char c)
{
  int dist;

  if ( !ptrHead )
    return NULL;
  if ( !ptrCurr )
    ptrCurr = ptrHead;

  /* Start searching either from the head of the list or from current hub,
   * whichever is closer to the character's hub */
  dist = c - ptrCurr->Key;
  if ( dist < 0 )
    dist = -dist;
  if ( c < dist )
    ptrCurr = ptrHead;

  if ( c < ptrCurr->Key ) /* Search backward */
    while ( ptrCurr && ptrCurr->Key != c )
      ptrCurr = ptrCurr->ptrPrev;
  else
    while ( ptrCurr && ptrCurr->Key != c )
      ptrCurr = ptrCurr->ptrNext; /* Search forward */

  return ptrCurr;
}


void FreeListMem(litemptr ptrHead)
```

```
{
  litemptr ptrTemp;

  while ( ptrHead ) {
    ptrTemp = ptrHead->ptrNext;
    free((void *) ptrHead);
    ptrHead = ptrTemp;
  }
}



/*
 * Local variables:
 * compile-command: "gcc 4_05.c && ./a.out 4_05.in"
 * End:
 */
```

```
/*
 * Задание 4.16.  Пусть L означает кольцевой двунаправленный список с заглавным
 * звеном.  Описать функцию или процедуру, которая из списка L, содержащего не
 * менее двух элементов, удаляет все элементы, у которых одинаковые "соседи"
 * (первый и последний элементы считать соседями).
 */

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

typedef struct _litem {
  int Key;
  int Value;
  struct _litem *ptrPrev;
  struct _litem *ptrNext;
} litem, *litemptr;

litemptr MakeUpList();
void     PrintOutList(litemptr ptrHead);
void     FreeListMem(litemptr ptrHead);


const char *strErrList ="error: list construction failed\n";


int main()
{
  litemptr ptrHead, ptrCurr, ptrTemp;
  int removed;

  /* Make up the list */
  if ( !(ptrHead = MakeUpList()) ) {
    fputs(strErrList, stderr);
    return -1;
  }

  /* Go through the ring list, removing items with identical
   * neighbors; stop as soon as no item is removed throughout one
   * whole cycle */
  ptrCurr = ptrHead;
  do {
    removed = 0;
    do {
      if ( ptrCurr->ptrPrev->Value == ptrCurr->ptrNext->Value &&
           ptrCurr->ptrPrev != ptrCurr->ptrNext ) {
        /* Cut out current item from the list */
        ptrCurr->ptrPrev->ptrNext = ptrCurr->ptrNext;
        ptrCurr->ptrNext->ptrPrev = ptrCurr->ptrPrev;
        /* Free memory taken up by it */
        ptrTemp = ptrCurr->ptrNext;
        free( (void *) ptrCurr );
        /* If what we are deleting is list head, change ptrHead to
         * point to something in the list */
        if ( ptrCurr == ptrHead )
          ptrHead = ptrTemp;
        ptrCurr = ptrTemp;
        removed = -1;
      } else
        ptrCurr = ptrCurr->ptrNext;
```

```
    } while ( ptrCurr != ptrHead );
  } while ( removed );

  /* Print out the resulting list */
  puts("Squeezed list:");
  PrintOutList(ptrHead);

  /* Free the list memory */
  FreeListMem(ptrHead);

  /* Bail out */
  return 0;
}


litemptr MakeUpList()
{
  int i, val;
  litemptr ptrHead, ptrPrev, ptrCurr;

  fputs("Enter the list (Ctrl-D to finish):\n", stdout);

  i = 0;
  if ( scanf("%d", &val) != EOF ) {
    ptrHead = (litemptr) malloc(sizeof(litem));
    if ( !ptrHead )
      return NULL;
    ptrHead->Key   = i++;
    ptrHead->Value = val;
    ptrPrev = ptrHead;

    while ( scanf("%d", &val) != EOF ) {
      ptrCurr = (litemptr) malloc(sizeof(litem));
      if ( !ptrCurr ) {
        ptrPrev->ptrNext = NULL;
        FreeListMem(ptrHead);
        return NULL;
      }
      ptrCurr->Key   = i++;
      ptrCurr->Value = val;
      ptrCurr->ptrPrev   = ptrPrev;
      ptrPrev->ptrNext   = ptrCurr;
      ptrPrev = ptrCurr;
    }
  } else
    return NULL;

  if ( i < 1 ){
    ptrPrev->ptrNext = NULL;
    FreeListMem(ptrHead);
    return NULL;
  }

  ptrPrev->ptrNext = ptrHead;
  ptrHead->ptrPrev = ptrPrev;

  return ptrHead;
}
```

```
void PrintOutList(litemptr ptrHead)
{
  litemptr ptrCurr;

  if ( !ptrHead )
    return;

  ptrCurr = ptrHead;
  do {
    printf("%d ", ptrCurr->Value);
    ptrCurr = ptrCurr->ptrNext;
  } while ( ptrCurr && ptrCurr != ptrHead );
  puts("");
}

void FreeListMem(litemptr ptrHead)
{
  litemptr ptrTemp, ptrCurr;

  ptrCurr = ptrHead;
  while ( ptrCurr && ptrCurr != ptrHead ) {
    ptrTemp = ptrCurr->ptrNext;
    free((void *) ptrCurr);
    ptrCurr = ptrTemp;
  }
}



/*
 * Local variables:
 * compile-command: "gcc 4_16.c && ./a.out <<< \"12 53 12 43 12\""
 * End:
 */
```

```
/*
 * Задание 5.5.  Удалить все узлы, имеющие одного сына.
 */

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

typedef struct _tnode {
  int Value;
  struct _tnode *ptrLeft;
  struct _tnode *ptrRight;
} tnode, *tnodeptr;

int  MakeUpTree(tnodeptr ptrRoot, int level);
int  CropTree(tnodeptr ptrRoot);
void PrintOutTree(tnodeptr ptrRoot);
void _PrintOutTree(tnodeptr ptrRoot, int level, char LR);
void FreeTreeMem(tnodeptr ptrRoot);
void SpaceOut(int level);


const char *strErrTree ="error: tree construction failed\n";


int main()
{
  tnodeptr ptrRoot, ptrCurr, ptrTemp;
  int removed;

  /* Seed the random number generator */
  srand(time(0));

  if ( !(ptrRoot = (tnodeptr) malloc(sizeof(tnode))) ) {
    fputs(strErrTree, stderr);
    return -1;
  }

  /* Make up the tree */
  if ( !MakeUpTree(ptrRoot, 0) ) {
    fputs(strErrTree, stderr);
    return -1;
  }

  if ( CropTree(ptrRoot) )
    ptrRoot = NULL;

  /* Print out the resulting tree */
  puts("\nCropped tree:");
  PrintOutTree(ptrRoot);

  /* Free the list memory */
  FreeTreeMem(ptrRoot);

  /* Bail out */
  return 0;
}


int MakeUpTree(tnodeptr ptrRoot, int level)
```

```
{
  int c, answer, i;

  if ( !ptrRoot )
    return 0;

  /* Make sure the values are sane */
  ptrRoot->Value = rand() / (RAND_MAX/10);

  /* Left branch */
  while ( -1 ) {
    SpaceOut(level);
    fprintf(stdout, "L%d, go left  (y/n)? ", level);
    answer = c = getchar();
    /* Discard remaining characters */
    while ( c != '\n' && c != EOF )
      c = getchar();

    if ( answer == 'y' ) {
      ptrRoot->ptrLeft = (tnodeptr) malloc( sizeof(tnode) );
      if ( !MakeUpTree(ptrRoot->ptrLeft, level+1) )
        return 0;
      break;
    } else if ( answer == 'n' )
      break;
  }

  /* Right branch */
  while ( -1 ) {
    SpaceOut(level);
    fprintf(stdout, "L%d, go right (y/n)? ", level);
    answer = c = getchar();
    /* Discard remaining characters */
    while ( c != '\n' && c != EOF )
      c = getchar();

    if ( answer == 'y' ) {
      ptrRoot->ptrRight = (tnodeptr) malloc( sizeof(tnode) );
      if ( !MakeUpTree(ptrRoot->ptrRight, level+1) )
        return 0;
      break;
    } else if ( answer == 'n' )
      break;
  }

  return -1;
}


/* Returns -1 if subtree starting with ptrRoot node was removed, 0 if
 * it remained intact */
int CropTree(tnodeptr ptrRoot)
{
  if ( !ptrRoot )
    return 0;

  /* Remove subtree starting with ptrRoot node if this node has only
   * one son */
  if ( (!ptrRoot->ptrLeft &&  ptrRoot->ptrRight) ||
```

```
       ( ptrRoot->ptrLeft && !ptrRoot->ptrRight) ) {
    FreeTreeMem(ptrRoot);
    return -1;
  }

  /* See if left or right subtrees should be removed, and if so,
   * cancel references to the removed subtrees from ptrRoot node */
  if ( CropTree(ptrRoot->ptrLeft) )
    ptrRoot->ptrLeft = NULL;
  if ( CropTree(ptrRoot->ptrRight) )
    ptrRoot->ptrRight = NULL;

  return 0;
}

void PrintOutTree(tnodeptr ptrRoot)
{
  _PrintOutTree(ptrRoot, 0, 'O');
}

void _PrintOutTree(tnodeptr ptrRoot, int level, char LR)
{
  if ( !ptrRoot )
    return;

  SpaceOut(level);
  printf("%c%d: %d\n", LR, level, ptrRoot->Value);
  if (ptrRoot->ptrLeft)
    _PrintOutTree(ptrRoot->ptrLeft, level+1, 'L');
  if (ptrRoot->ptrRight)
    _PrintOutTree(ptrRoot->ptrRight, level+1, 'R');
}

void FreeTreeMem(tnodeptr ptrRoot)
{
  if (!ptrRoot)
    return;

  FreeTreeMem(ptrRoot->ptrLeft);
  FreeTreeMem(ptrRoot->ptrRight);
  free((void *) ptrRoot);
}

void SpaceOut(int level)
{
  int i;

  for ( i = 0; i < level; i++ )
    putchar(' ');
}



/*
 * Local variables:
 * compile-command: "gcc 5_05.c"
 * End:
 */
```

```
/*
 * Задание 6.5.  Даны числа A_1, A_2, ..., A_n.  Получить в порядке возрастания
 * все различные числа, входящие в A_1, A_2, ..., A_n.
 */

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

#define IN_BUFFER_LEN           20

typedef struct _litem {
  int value;
  struct _litem *ptrNext;
} litem, *litemptr;

int  iout(int n, int pos);
void PrintOutList(litemptr ptrHead);
void FreeListMem(litemptr ptrHead);

int main(void)
{
  int i, *InBuf, ret = 0;
  litemptr ptrHead, ptrNew, ptrTemp;

  /* Allocate and fill in the array */
  if ( !( InBuf = (int *) malloc(IN_BUFFER_LEN*sizeof(*InBuf)) ) )
    return -1;
  srand(time(0));
  for ( i = 0; i < IN_BUFFER_LEN; i++ )
    *(InBuf + i) = rand() / (RAND_MAX / 10);

  /* Print out array */
  for ( i = 0; i < IN_BUFFER_LEN; i++ )
    printf("%d ", *(InBuf + i));

  putchar('\n');

  /* Allocate memory for the list head */
  if ( !(ptrHead = (litemptr) malloc(sizeof(litem))) )
    return -1;

  ptrHead->value = *InBuf;
  ptrHead->ptrNext = NULL;

  for ( i = 1; i < IN_BUFFER_LEN; i++ ) {
    int arr_el = *(InBuf + i);

    if ( arr_el < ptrHead->value ) {
      /* Add new list item before the head */
      if ( !(ptrNew = (litemptr) malloc(sizeof(litem))) ) {
        ret = -1;
        break;
      }
      ptrNew->value = arr_el;
      ptrNew->ptrNext = ptrHead;
      ptrHead = ptrNew;
    } else if ( arr_el > ptrHead->value ) { /* Need this IF to get rid of
                                               'arr_el == ptrHead->value' situation */
      ptrTemp = ptrHead;
```

```
      /* Search through the list until we reach the end or until we see that
         next item is greater than arr_el */
      while ( ptrTemp->ptrNext && ptrTemp->ptrNext->value < arr_el )
        ptrTemp = ptrTemp->ptrNext;
      /* Add new list item if we reached the end of list or next item is not
         equal to arr_el (i.e. greater than arr_el) */
      if ( !ptrTemp->ptrNext || ptrTemp->ptrNext->value != arr_el ) {
        if ( !(ptrNew = (litemptr) malloc(sizeof(litem))) ) {
          ret = -1;
          break;
        }
        ptrNew->value = arr_el;
        ptrNew->ptrNext = ptrTemp->ptrNext;
        ptrTemp->ptrNext = ptrNew;
      }
    }
  }

  PrintOutList(ptrHead);

  FreeListMem(ptrHead);

  return ret;
}

void PrintOutList(litemptr ptrHead)
{
  while ( ptrHead ) {
    printf("%d ", ptrHead->value);
    ptrHead = ptrHead->ptrNext;
  }

  putchar('\n');

  return;
}

void FreeListMem(litemptr ptrHead)
{
  litemptr ptrNext;

  while ( ptrHead ) {
    ptrNext = ptrHead->ptrNext;
    free( (void *) ptrHead );
    ptrHead = ptrNext;
  }

  return;
}



/*
 * Local variables:
 * compile-command: "gcc 6_05.c && echo && ./a.out"
 * End:
 */
```

```
/*
 * Задание 6.16.  Дана действительная матрица размером MxN; упорядочить
 * (переставить) столбцы матрицы по невозрастанию сумм элементов столбцов.
 * Разрешается использовать дополнительный массив.
 */

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

#define M       10
#define N       3

double **matrix;

void MatrixOut(void);

int main(void)
{
  int i, j, k, gotcha;
  double curr;
  double *ptrTemp;


  /* Seed the pseudo-number generator */
  srand(time(0));
  if ( !(matrix = (double **) malloc(sizeof(double *) * M)) )
    return -1;
  /* Allocate space for and fill in the matrix (last element in each row will
     hold the sum of that row) */
  for ( i = 0; i < M; i++) {
    if ( !( ptrTemp = (double *) malloc(sizeof(double) * (N + 1)) ) )
      return -1;
    *(matrix + i) = ptrTemp;
    for ( j = 0; j < N; j++ )
      *(ptrTemp + j) = (double) ( rand() / (RAND_MAX / 10) );
  }

  /* Calculate column sums and display matrix and sums */
  for ( i = 0; i < M; i++ ) {
    ptrTemp = *(matrix + i);
    for ( j = 0; j < N; j++)
      *(ptrTemp + N) += *(ptrTemp + j);
  }
  fputs("Original matrix:\n\n", stdout);
  MatrixOut();

  /* Sort the rows in non-ascending order */
  for ( i = M - 1, gotcha = -1; i > 0 && gotcha; i-- ) {
    gotcha = 0;
    /* Bubble up the row with largest sum */
    for ( j = 0; j < i; j++ ) {
      k = j++;
      curr = *(*(matrix + k) + N);
      /* Scan rows until current sum is >= than consequent ones */
      while ( j <= i && curr >= *(*(matrix + j) + N) )
        j++;
      j--; /* Return back */
      if ( k != j ) {
        /* Swap the rows */
```

```
        ptrTemp = *(matrix + k);
        *(matrix + k) = *(matrix + j);
        *(matrix + j) = ptrTemp;
        gotcha = -1;
      }
    }
  }

  fputs("\n\nSorted matrix:\n\n", stdout);
  MatrixOut();
  putchar('\n');

  return 0;
}

void MatrixOut(void)
{
  int i, j;

  /* Print the transposed matrix */
  for ( j = 0; j < N; j++ ) {
    for ( i = 0; i < M; i++)
      printf("%3.0lf", *(*(matrix + i) + j));
    putchar('\n');
  }
  for ( j = 0; j < 3 * M; j++ )
    putchar('-');
  putchar('\n');
  /* Print the sums */
  for ( i = 0; i < M; i++)
    printf("%3.0lf", *(*(matrix + i) + N));
}



/*
 * Local variables:
 * compile-command: "gcc 6_16.c && echo && ./a.out"
 * End:
 */
```

```
/*
 * Задание 7.5.  Пусть A и B--файлы, k--натуральное число.  Будем говорить, что
 * файлы A и B согласованно k-упорядочены, если
 * 1) в каждом из фалов A и B первые k компонент, следующие за ними k компонент
 *    и т.д. образуют упорядоченные группы; последняя группа файла (тоже
 *    упорядоченная) может быть неполной, т.е. содержать менее k компонент, но
 *    при этом только один из файлов A и B может иметь неполную последнюю
 *    группу;
 * 2) число упорядоченных групп файла A отличается от числа упорядоченных групп
 *    файла B не более чем на единицу;
 * 3) если в одном файле число упорядоченных групп меньше на 1, чем в другом, то
 *    неполной может быть только посленяя группа более длинного файла.
 *
 * Компоненты двух согласованно k-упорядоченных файлов разместить в файлах g и h
 * так, что они будут согласованно 2k-упорядочены.
 */

#include <stdio.h>

const char *FileNameIn1  = "7_05.1.in";
const char *FileNameIn2  = "7_05.2.in";
const char *FileNameOut1 = "7_05.1.out";
const char *FileNameOut2 = "7_05.2.out";
long k = 5;                                     /* length of k-groups */


int main(void)
{
  FILE *fin1, *fin2, *fout1, *fout2, *ftemp;
  int   val1, val2;                             /* vars to read stuff into */
  int   val1_read, val2_read;                   /* read error flags */
  long  count1, count2;                         /* count of output elements */
  long  two_k = k + k;                          /* length of 2k-group */

  /* Try and open the files */
  if ( !(fin1 = fopen(FileNameIn1,"r")) || !(fin2 = fopen(FileNameIn2,"r")) )
    return -1;
  if ( !(fout1 = fopen(FileNameOut1,"w")) || !(fout2 = fopen(FileNameOut2,"w"))
 )
    return -1;

  val1_read = fscanf(fin1, "%d", &val1);
  val2_read = fscanf(fin2, "%d", &val2);

  while ( val1_read != EOF || val2_read != EOF ) {
    count1 = count2 = 0;

    /* Output sorted 2k-group to out-file No1 by mixing one k-group from 1st
       in-file with another k-group from 2nd in-file  */
    while ( ( val1_read != EOF || val2_read != EOF ) &&   /* anything left... */
            count1 + count2 < two_k ) {  /* ... and 2k-group not finished yet */

      if ( ( val2_read == EOF ||                /* 2nd file is done for */
             count2 >= k      ||                /* k-group from 2nd file done */
             val1 <= val2 && count1 < k ) &&    /* sorted order */
           val1_read != EOF ) {                 /* anything been read */
        fprintf(fout1, "%d\n", val1);
        count1++;
        val1_read = fscanf(fin1, "%d", &val1);
```

```
        continue;
      }

      if ( ( val1_read == EOF ||                /* 1st file is done for */
             count1 >= k      ||                /* k-group from 1st file done */
             val2 <= val1 && count2 < k ) &&    /* sorted order */
           val2_read != EOF ) {                 /* anything been read */
        fprintf(fout1, "%d\n", val2);
        count2++;
        val2_read = fscanf(fin2, "%d", &val2);
      }
    }

    /* Separate 2k-groups by a newline */
    putc('\n', fout1);
    /* Switch to the other file for next 2k-group output */
    ftemp = fout1, fout1 = fout2, fout2 = ftemp;
  }

  fclose(fin1);
  fclose(fin2);
  fclose(fout1);
  fclose(fout2);

  return 0;
}



/*
 * Local variables:
 * compile-command: "gcc 7_05.c && echo && ./a.out"
 * End:
 */
```

```
/*
 * Задание 7.16.  Составить программу, помогающую сотрудникам ГАИ.  В файле
 * хранится информация об автомобилях:  регистрационный номер автомобиля, цвет
 * автомобиля, марка, год выпуска, адрес проживания владельца.  По требованию
 * выдавать сведения об автолюбителях, имеющих:
 *   - автомобиль с заданным регистрационным номером;
 *   - автомобиль заданного цвета;
 *   - автомобиль заданной марки.
 */

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

#define LINE_BUFFER_LEN         256
#define NUM_FIELDS              5
#define NUM_FILTERED_FIELDS     3

const char *strUsageShort     = "Usage: %s [OPTION]... [FILE]\n\
Try `%s -h' for more information.\n";

const char *strUsageLong      = "Usage: %s [OPTION]... [FILE]\n\
Filter records from FILE (STDIN if no filename given) and print them on STDOUT.\n\
Example: %s -i -m '^ford' -c black car.db\n\
\n\
Filter selection and interpretation:\n\
  -i          toggle ignore case distinctions; can be supplied more than\n\
              once, toggles case distinctions for all subsequent filters;\n\
              initially case is honored\n\
  -p REGEX    add regular expression REGEX as car license plate filter\n\
  -c REGEX    add regular expression REGEX as car color filter\n\
  -m REGEX    add regular expression REGEX as car model filter\n\
\n\
Miscellaneous:\n\
  -h    display this help and exit\n\
\n\
Filter combinations:\n\
  More than one filter of the same type can be specified; in that case, all\n\
  the filters of the same type are OR'ed together.  For instance, options\n\
  `-c ^bl -c red' will select all cars of colors beginning with `bl' or\n\
  containing `red'.\n\
\n\
  Filters for different fields are AND'ed together.  For instance, options\n\
  `-i -c '^black$' -m ford' will select all black fords.\n\
\n\
Database file format:\n\
  Field 1: number on the license plate of the car\n\
  Field 2: color of the car\n\
  Field 3: brand and model of the car\n\
  Field 4: year of production of the car\n\
  Field 5: address of the owner of the car\n\
\n\
  Each field starts on a new line; one empty line separates records.\n";

const char *strOptions        = ":hip:c:m:";
const char *strErrOpenFile    = "%s: error opening file `%s'.\n";
const char *strMissingOptArg = "%s: option `-%c' requires an argument <%s>.\n";
const char *strUnknownOpt     = "%s: unknown option `-%c'.\n";
```

```
const char *strUnknownOptChar= "%s: unknown option character '\\x%x'.\n";
const char *strErrMem        = "%s: not enough memory.\n";
const char *strErrUnexpectedEOF = "%s: error reading input--unexpected end of file.\n";
      char *strProgName;

int RegexIgnoreCase = 0; /* initially honor the case */

typedef struct _filter {
  int       len;
  regex_t **filter;
} filter, *filterptr;

filterptr Filters;
char    **Fields;


int  SetUpBuffers(void);
int  ParseOptions(int argc, char **argv);
void PrintRegError(int RegErr, regex_t *compiled);


int main(int argc, char **argv)
{
  int   i, j;
  int   BailOut;
  char *ptrTemp;

  strProgName = *argv;

  if ( !SetUpBuffers() )
    return -1;

  if ( !ParseOptions(argc, argv) )
    return -1;

  /* Main loop */
  BailOut = 0;
  while ( !BailOut ) {
    int reject = 0;

    /* Read in all the fields of the current record */
    for ( i = 0; i < NUM_FIELDS; i++ ) {
      if ( !fgets(*(Fields + i), LINE_BUFFER_LEN, stdin) ) {
        if ( i != 0 )
          fprintf(stderr, strErrUnexpectedEOF, strProgName);
        BailOut = -1;
        break;
      }
      /* Remove trailing newline from the buffer */
      ptrTemp = *(Fields + i);
      while ( *ptrTemp )
        ptrTemp++;
      ptrTemp--;
      if ( *ptrTemp == '\n' )
        *ptrTemp = '\0';
    }
    /* Filter this record */
    if ( !BailOut ) {
      /* Check record against the filter */
```

```
      for ( i = 0; i < NUM_FILTERED_FIELDS; i++ ) {
        regex_t **ptrFilter = (Filters + i)->filter;
        int       satisfied = 0;
        for ( j = (Filters + i)->len - 1; j >= 0; j-- )
          if ( !regexec(*(ptrFilter + j), *(Fields + i), 0, NULL, 0) ) {
            satisfied = -1;
            break;
          }
        if ( !satisfied && (Filters + i)->len > 0 ) {
          reject = -1;
          break;
        }
      }
      /* Print the record if it passed through the filter */
      if ( !reject ) {
        for ( i = 0; i < NUM_FIELDS; i++ )
          printf("%s\n", *(Fields + i));
        putchar('\n');
      }
      /* Read in newline separating the records */
      fgets(*Fields, LINE_BUFFER_LEN, stdin);
    }
  }

  return 0;
}


int SetUpBuffers(void)
{
  int i;

  if ( !(Filters = malloc(NUM_FILTERED_FIELDS * sizeof( *Filters ))) ||
       !(Fields  = malloc(NUM_FIELDS          * sizeof( *Fields  ))) ) {
    fprintf(stderr, strErrMem, strProgName);
    return 0;
  }

  for ( i = 0; i < NUM_FIELDS; i++ )
    if ( !(*(Fields+i) = malloc(LINE_BUFFER_LEN)) ) {
      fprintf(stderr, strErrMem, strProgName);
      return 0;
    }

  for ( i = 0; i < NUM_FILTERED_FIELDS; i++ ) {
    (Filters + i)->len = 0;
    (Filters + i)->filter = NULL;
  }

  return -1;
}


int ParseOptions(int argc, char **argv)
{
  int       c, i, len, RegErr;
  regex_t **ptrFilter;
  char      *ptrTemp;
```

```
  opterr = 0; /* prevent GETOPT from printing error messages */

  while ( (c = getopt (argc, argv, strOptions)) != -1 ) {
    switch ( c ) {
    case 'h': /* help */
      fprintf(stdout, strUsageLong, strProgName, strProgName);
      return 0;
    case 'i': /* toggle case insensitive regexp search */
      RegexIgnoreCase = (RegexIgnoreCase == 0) ? REG_ICASE : 0;
      continue;
      break;
    case 'p': /* license plate */
      i = 0;
      break;
    case 'c': /* color */
      i = 1;
      break;
    case 'm': /* model */
      i = 2;
      break;
    case '?': /* unknown option */
      if ( isprint (optopt) )
        fprintf(stderr, strUnknownOpt, strProgName, optopt);
      else
        fprintf(stderr, strUnknownOptChar, strProgName, optopt);
      fprintf(stderr, strUsageShort, strProgName, strProgName);
      return 0;
      break;
    case ':': /* missing option argument */
      switch ( optopt ) {
      case 'p':
        ptrTemp = "license plate";
        break;
      case 'c':
        ptrTemp = "color";
        break;
      case 'm':
        ptrTemp = "model";
        break;
      default:
        abort(); /* should never get here */
      }
      fprintf(stderr, strMissingOptArg, strProgName, optopt, ptrTemp);
      return 0;
      break;
    default:
      abort(); /* should never get here */
    }
    /* Add new regexp to the filter */
    len = ++(Filters+i)->len;
    ptrFilter = (Filters+i)->filter;
    /* Grow filter array by one and allocate new regex structure */
    if ( !(ptrFilter = (regex_t **)
           realloc(ptrFilter, len * sizeof(*ptrFilter))) ||
         !(*(ptrFilter + len - 1) = (regex_t *) malloc(sizeof(regex_t))) ) {
      fprintf(stderr, strErrMem, strProgName);
      return 0;
    }
    (Filters+i)->filter = ptrFilter; /* save new location of filter array */
```

```
    /* Compile the regex right off */
    RegErr=regcomp(*(ptrFilter + len - 1), optarg, REG_NOSUB | RegexIgnoreCase);
    if ( RegErr ) {
      PrintRegError(RegErr, *(ptrFilter+len-1));
      return 0;
    }
  }

  /* If user provided filename, try to redirect stdin to that file */
  if ( optind < argc ) {
    if ( !freopen(*(argv+optind), "r", stdin) ) {
      fprintf(stderr, strErrOpenFile, strProgName, *(argv+optind));
      return 0;
    }
  }

  return -1;
}


void PrintRegError(int RegErr, regex_t *compiled)
{
  char *strErr;
  int len;

  len = regerror(RegErr, compiled, NULL, 0);
  if ( !(strErr = (char *) malloc(len)) ) {
    fprintf(stderr, strErrMem, strProgName);
    return;
  }
  regerror(RegErr, compiled, strErr, len);
  fputs(strProgName, stderr);
  fputc(':', stderr);
  fputc(' ', stderr);
  fputs(strErr, stderr);
  fputc('\n', stderr);
  free((void *) strErr);
  return;
}



/*
 * Local variables:
 * compile-command: "gcc 7_16.c && echo && ./a.out -c '^bl' -c red -i -m ford 7_
16.in"
 * End:
 */
```

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

#define T_S(idx)    (*(ttable + idx * 2))
#define T_E(idx)    (*(ttable + idx * 2 + 1))

int n;
int *ttable;
int *selected;
int max;

int main(void)
{
  int i, j, count;

  /* Enter time-table length */
  scanf("%d", &n);

  /* Allocate buffers */
  ttable   = (int *) malloc( sizeof(int) * n * 2 );
  selected = (int *) malloc( sizeof(int) * n );

  /* Enter time-table */
  for ( i = 0; i < n; i++ )
    scanf("%d%d", ttable + i + i, ttable + i + i + 1);

  /* Main loop */
  max   = 0;
  count = 0;
  j = 0;
  do {
    if ( j == n || count > n ) {
      if ( count > max )
        max = count;
      if ( count == n )
        break;
      j = *(selected + --count);
      //printf("- %d %d\n", j, count);
    } else {
      for ( i = 0; i < count; i++ )
        if ( !(T_S(j) < T_S(*(selected+i)) && T_E(j) < T_S(*(selected+i))) &&
             !(T_S(j) > T_E(*(selected+i)) && T_E(j) > T_E(*(selected+i))) )
          break;
      if ( i == count ) {
        *(selected + count++) = j;
        //printf("+ %d %d\n", j, count);
        j = 0;
        continue;
      }
    }
    j++;
  } while ( count >= 0 );

  printf("%d\n", max);

  return 0;
}
```

```
/*
 * Local variables:
 * compile-command: "gcc -Wall f.c && ./a.out <<< \"5  3 4  1 5  6 7  4 5  1 3\"
"
 * End:
 */
```

```
/*
 * Prints out all combinations of 0's and 1's in a binary set of length N
 */
#include <stdio.h>

#define N       5

int bset[N];

void print_bset(void);

int main(void)
{
  int i;

  for ( i = 0; i < N; i++ )
    bset[i] = 0;

  print_bset();

  /* bset is treated as binary set, we "add" 1 to it to get next combination */
  while ( 1 ) {
    for ( i = 0; i < N && bset[i] == 1; i++ )
      bset[i] = 0;
    if ( i == N )
      break;
    bset[i] = 1;
    print_bset();
  }

  return 0;
}

void print_bset(void)
{
  int i;

  for ( i = N - 1; i >= 0; i-- )
    printf("%d ", bset[i]);
  putchar('\n');
}



/*
 * Local variables:
 * compile-command: "gcc -Wparentheses -W combinations.c && ./a.out"
 * End:
 */
```

```
/*
 * Prints out all decompositions of an integer into its addends.
 */

#include <stdio.h>

void addend_decomp(int n, int l);
void space_out(int l);

int main(void)
{
  int n;

  fputs("Enter N: ", stdout);
  scanf("%d", &n);

  addend_decomp(n, 0);

  return 0;
}

void addend_decomp(int n, int l)
{
  int i;

  if ( n == 1 ) {
    fputc('1', stdout);
    fputc('\n', stdout);
    return;
  }

  fprintf(stdout, "%d\n", n);
  for ( i = 1; i < n; i++ ) {
    space_out(l);
    fprintf(stdout, "%d+", n-i);
    addend_decomp(i, l+1);
  }

  return;
}

void space_out(int l)
{
  int i;

  for ( i = l * 2; i > 0; i--)
    fputc(' ', stdout);

  return;
}



/*
 * Local variables:
 * compile-command: "gcc recursive_addend_decomp.c && echo && ./a.out <<< 5"
 * End:
 */
```

```c
/*
 * Prints out all combinations of 0's and 1's in a binary set of length N
 */
#include <stdio.h>

#define N       6

int bset[N];

void combination(int i);

int main(void)
{
  combination(0);

  return 0;
}


void combination(int i)
{
  int j;

  if ( i == N ) {                    /* no more elements in array to add */
    for ( j = 0; j < N; j++)
      printf("%d ", bset[j]);
    putchar('\n');

    return;
  }

  bset[i] = 0;
  combination(i + 1);
  bset[i] = 1;
  combination(i + 1);

  return;
}



/*
 * Local variables:
 * compile-command: "gcc -Wall recursive_combinations.c && ./a.out"
 * End:
 */
```

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


const char *strErrOpenFile       = "error: cannont open file '%s'\n";
const char *strErrMem            = "error: not enough memory\n";
const char *strErrUnexpectedEOF = "error: unexpected end of input\n";

int      n;
double **matrix;


double determ(int l, double **matrix);


int main(int argc, char **argv)
{
  int i, j;
  int interact = -1;

  /* If user provided filename, try to redirect stdin to that file */
  if ( argc > 1 ) {
    if ( !freopen(*(argv+1), "r", stdin) ) {
      fprintf(stderr, strErrOpenFile, *(argv+1));
      return -1;
    }
    interact = 0;
  }

  /* Get matrix dimensions from user */
  if ( interact )
    fputs("Enter matrix dimension: ", stdout);
  if ( scanf("%d", &n) == EOF ) {
    fputs(strErrUnexpectedEOF, stderr);
    return -1;
  }

  /* Memory allocations */
  if ( !( matrix = (double **) malloc( sizeof(double *) * n ) ) ) {
    fputs(strErrMem, stderr);
    return -1;
  }

  for ( i = 0; i < n; i++ ) {
    if ( !( *(matrix + i) = (double *) malloc( sizeof(double) * n) ) ) {
      fputs(strErrMem, stderr);
      return -1;
    }
  }

  /* Get matrix elements from user */
  if ( interact )
    fputs("Enter matrix elements: ", stdout);

  for ( i = 0; i < n; i++ )
    for ( j = 0; j < n; j++ )
      if ( scanf("%lf", *(matrix + i) + j) == EOF ) {
        fputs(strErrUnexpectedEOF, stderr);
        return -1;
```

```
      }

  /* Print out the matrix */
  for ( i = 0; i < n; i++ ) {
    for ( j = 0; j < n; j++ )
      fprintf(stdout, "%6.2lf ", *(*(matrix + i) + j));
    fputc('\n', stdout);
  }

  /* Calculate and print determinant */
  fprintf(stdout, "\n%lf\n", determ(0, matrix));

  /* Free the mallocs */
  for ( i = 0; i < n; i++ )
    free((void *) *(matrix + i));
  free((void *) matrix);

  return 0;
}


/*
 * Calculates determinant by adding up algebraic complements for the first
 * column.  Recursively calls itself to calculate determinants for the algebraic
 * complements.
 */
double determ(int l, double **matrix)
{
  int      i;
  double  *ptrTemp;
  double   sum        = 0;
  int      l_plus_one = l + 1;
  int      n_minus_l  = n - l;
  double **ptrNext    = matrix + 1;

  if ( l_plus_one == n )
    return *(*matrix + l);

  sum += *(*matrix + l) * determ(l_plus_one, ptrNext);
  for ( i = 1; i < n_minus_l; i++ ) {
    /* Swap i-th line with the first line */
    ptrTemp = *matrix;
    *matrix = *(matrix + i);
    *(matrix + i) = ptrTemp;

    /* Swapping first line with i-th line automagically makes sign of this
       complement a minus:  number of transpositions required to exchange first
       line with i-th line is even for even i's (minus remains) and odd for odd
       i's (plus is negated) */
    sum -= *(*matrix + l) * determ(l_plus_one, ptrNext);

    /* Swap first line back into place */
    ptrTemp = *matrix;
    *matrix = *(matrix + i);
    *(matrix + i) = ptrTemp;
  }

  return sum;
}
```

```
/*
 * Local variables:
 * compile-command: "gcc recursive_determinant.c && echo && ./a.out <<< \"3 1 -1
 0 -1 2 3 -1 0 1\""
 * End:
 */
```

```
/*
 * Prints out all permutations of integers in the range 0 to (n - 1) inclusive
 */

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

int *ptrHead, n;

void permutation(int i);

int main(void)
{
  int i;

  fputs("Enter N: ", stdout);
  scanf("%d", &n);
  if ( !(ptrHead = (int *) malloc (n * sizeof(*ptrHead))) )
       return -1;
  for ( i = 0; i < n; i++ )
    *(ptrHead + i) = i;

  permutation(0);

  return 0;
}

void permutation(int i)
{
  int j, temp, i_plus_one = i + 1;

  /* If we reached the end of the array, print out next permutation */
  if ( i == n - 1 ) {
    for ( j = 0; j < n; j++ )
      printf("%d ", *(ptrHead+j));
    putchar('\n');

    return;
  }

  /* Switch first element with each consecutive element, calling recursively
     permutation function for each of the combinations (after each call switch
     the elements back into place) */
  permutation(i_plus_one);
  for ( j = i_plus_one; j < n; j++ ) {
    temp = *(ptrHead+i); *(ptrHead+i) = *(ptrHead+j); *(ptrHead+j) = temp;
    permutation(i_plus_one);
    temp = *(ptrHead+i); *(ptrHead+i) = *(ptrHead+j); *(ptrHead+j) = temp;
  }

  return;
}


/*
 * Local variables:
 * compile-command: "gcc recursive_permutations.c && echo && ./a.out <<< 5"
 * End:
 */
```