SECOND EDITION

# THE C ANSWER BOOK

Solutions to the Exercises in
**The C Programming Language**, second edition
by Brian W. Kernighan & Dennis M. Ritchie

*1. Run the "hello, world" program on your system. Experiment with leaving out parts of the program, to see what error messages you get.*

```
#include <stdio.h>

int main(void)
{
  printf("hello, world\n");
  return 0;
}
```

*2. Experiment to find out what happens when `printf` 's argument string contains \c, where c is some character not listed above.*

```
#include <stdio.h>

int main(void)
{
  printf("Audible or visual alert. \a\n");
  printf("Form feed. \f\n");
  printf("This escape, \r, moves the active position to the initial position of the current line.\n");
  printf("Vertical tab \v is tricky, as its behaviour is unspecified under certain conditions.\n");

  return 0;
}
```

*3. Modify the temperature conversion program to print a heading above the table.*

```
#include <stdio.h>

int main(void)
{
  float fahr, celsius;
  int lower, upper, step;

  lower = 0;
  upper = 300;
  step = 20;

  printf("F     C\n\n");
  fahr = lower;
  while(fahr <= upper)
```

```
  {
    celsius = (5.0 / 9.0) * (fahr - 32.0);
    printf("%3.0f %6.1f\n", fahr, celsius);
    fahr = fahr + step;
  }
  return 0;
}
```

4. *Write a program to print the corresponding Celsius to Fahrenheit table.*

```
#include <stdio.h>

int main(void)
{
  float fahr, celsius;
  int lower, upper, step;

  lower = 0;
  upper = 300;
  step = 20;

  printf("C      F\n\n");
  celsius = lower;
  while(celsius <= upper)
  {
    fahr = (9.0/5.0) * celsius + 32.0;
    printf("%3.0f %6.1f\n", celsius, fahr);
    celsius = celsius + step;
  }
  return 0;
}
```

5. *Modify the temperature conversion program to print the table in reverse order, that is, from 300 degrees to 0.*

```
#include <stdio.h>

int main(void)
{
  float fahr, celsius;
  int lower, upper, step;
```

```c
  lower = 0;
  upper = 300;
  step = 20;

  printf("C     F\n\n");
  celsius = upper;
  while(celsius >= lower)
  {
    fahr = (9.0/5.0) * celsius + 32.0;
    printf("%3.0f %6.1f\n", celsius, fahr);
    celsius = celsius - step;
  }
  return 0;
}
```

6. *Verify that the expression* `getchar() != EOF` *is 0 or 1.*

```c
#include <stdio.h>

int main(void)
{
  printf("Press a key. ENTER would be nice :-)\n\n");
  printf("The expression getchar() != EOF evaluates
to %d\n", getchar() != EOF);
  return 0;
}
```

7.

```c
#include <stdio.h>

int main(void)
{
  printf("The value of EOF is %d\n\n", EOF);

  return 0;
}
```

8. *Write a program to count blanks, tabs, and newlines.*

```c
#include <stdio.h>
int main(void)
{
  int blanks, tabs, newlines;
  int c;
  int done = 0;
```

```
  int lastchar = 0;

  blanks = 0;
  tabs = 0;
  newlines = 0;

  while(done == 0)
  {
    c = getchar();

    if(c == ' ')
      ++blanks;

    if(c == '\t')
      ++tabs;

    if(c == '\n')
      ++newlines;

    if(c == EOF)
    {
      if(lastchar != '\n')
      {
        ++newlines; /* this is a bit of a semantic
stretch, but it copes
                     * with implementations where a
text file might not
                     * end with a newline. Thanks to
Jim Stad for pointing
                     * this out.
                     */
      }
      done = 1;
    }
    lastchar = c;
  }

  printf("Blanks: %d\nTabs: %d\nLines: %d\n", blanks,
tabs, newlines);
  return 0;
}
```

9.*Write a program to copy its input to its output, replacing each string of one or more blanks by a single blank.*

```
include <stdio.h>

int main(void)
{
  int c;
  int inspace;

  inspace = 0;
  while((c = getchar()) != EOF)
  {
    if(c == ' ')
    {
      if(inspace == 0)
      {
        inspace = 1;
        putchar(c);
      }
    }

    /* We haven't met 'else' yet, so we have to be a
little clumsy */
    if(c != ' ')
    {
      inspace = 0;
      putchar(c);
    }
  }

  return 0;
}
```

10.*Write a program to copy its input to its output, replacing each tab by* `\t` *, each backspace by* `\b` *, and each backslash by* `\\` *. This makes tabs and backspaces visible in an unambiguous way.*

```
#include <stdio.h>

int main()
{
    int c, d;
```

```
    while ( (c=getchar()) != EOF) {
        d = 0;
        if (c == '\\') {
            putchar('\\');
            putchar('\\');
            d = 1;
        }
        if (c == '\t') {
            putchar('\\');
            putchar('t');
            d = 1;
        }
        if (c == '\b') {
            putchar('\\');
            putchar('b');
            d = 1;
        }
        if (d == 0)
            putchar(c);
    }
    return 0;
}
```

11. *How would you test the word count program? What kinds of input are most likely to uncover bugs if there are any?*

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

int main(void)
{
    FILE *f;
    unsigned long i;
    static char *ws = " \f\t\v";
    static char *al = "abcdefghijklmnopqrstuvwxyz";
    static char *i5 = "a b c d e f g h i j k l m "
                      "n o p q r s t u v w x y z "
                      "a b c d e f g h i j k l m "
                      "n o p q r s t u v w x y z "
                      "a b c d e f g h i j k l m "
                      "n\n";
```

```
    /* Generate the following: */
    /* 0. input file contains zero words */
    f = fopen("test0", "w");
    assert(f != NULL);
    fclose(f);

    /* 1. input file contains 1 enormous word without
any newlines */
    f = fopen("test1", "w");
    assert(f != NULL);
    for (i = 0; i < ((66000ul / 26) + 1); i++)
        fputs(al, f);
    fclose(f);

    /* 2. input file contains all white space without
newlines */
    f = fopen("test2", "w");
    assert(f != NULL);
    for (i = 0; i < ((66000ul / 4) + 1); i++)
        fputs(ws, f);
    fclose(f);

    /* 3. input file contains 66000 newlines */
    f = fopen("test3", "w");
    assert(f != NULL);
    for (i = 0; i < 66000; i++)
        fputc('\n', f);
    fclose(f);

    /* 4. input file contains word/
     *    {huge sequence of whitespace of different
kinds}
     *    /word
     */
    f = fopen("test4", "w");
    assert(f != NULL);
    fputs("word", f);
    for (i = 0; i < ((66000ul / 26) + 1); i++)
        fputs(ws, f);
    fputs("word", f);
```

```
    fclose(f);

    /* 5. input file contains 66000 single letter
words,
     *    66 to the line
     */
    f = fopen("test5", "w");
    assert(f != NULL);
    for (i = 0; i < 1000; i++)
        fputs(i5, f);
    fclose(f);

    /* 6. input file contains 66000 words without any
newlines */
    f = fopen("test6", "w");
    assert(f != NULL);
    for (i = 0; i < 66000; i++)
        fputs("word ", f);
    fclose(f);

    return 0;
}
```

12. *Write a program that prints its input one word per line.*

```
#include <stdio.h>

int main(void)
{
    int c;
    int inspace;

    inspace = 1;
    while ((c = getchar()) != EOF) {
        if (c == ' ' || c == '\t' || c == '\n') {
            if (inspace == 0) {
                inspace = 1;
                putchar('\n');
            }
            /* else, don't print anything */
        } else {
            inspace = 0;
            putchar(c);
```

```
        }
    }

    return 0;
}
```

13.*Write a program to print a histogram of the lengths of words in its input. It is easy to draw the histogram with the bars horizontal; a vertical orientation is more challenging.*

```
#include <stdio.h>

#define MAXWORDLEN 10

int main(void)
{
  int c;
  int inspace = 0;
  long lengtharr[MAXWORDLEN + 1];
  int wordlen = 0;

  int firstletter = 1;
  long thisval = 0;
  long maxval = 0;
  int thisidx = 0;
  int done = 0;

  for(thisidx = 0; thisidx <= MAXWORDLEN; thisidx++)
  {
    lengtharr[thisidx] = 0;
  }

  while(done == 0)
  {
    c = getchar();

    if(c == ' ' || c == '\t' || c == '\n' || c ==
EOF)
    {
      if(inspace == 0)
      {
        firstletter = 0;
        inspace = 1;
```

```
      if(wordlen <= MAXWORDLEN)
      {
        if(wordlen > 0)
        {
          thisval = ++lengtharr[wordlen - 1];
          if(thisval > maxval)
          {
            maxval = thisval;
          }
        }
      }
      else
      {
        thisval = ++lengtharr[MAXWORDLEN];
        if(thisval > maxval)
        {
          maxval = thisval;
        }
      }
    }
    if(c == EOF)
    {
      done = 1;
    }
  }
  else
  {
    if(inspace == 1 || firstletter == 1)
    {
      wordlen = 0;
      firstletter = 0;
      inspace = 0;
    }
    ++wordlen;
  }
}

for(thisval = maxval; thisval > 0; thisval--)
{
  printf("%4d  | ", thisval);
```

```
    for(thisidx = 0; thisidx <= MAXWORDLEN; thisidx+
+)
    {
      if(lengtharr[thisidx] >= thisval)
      {
        printf("*  ");
      }
      else
      {
        printf("   ");
      }
    }
    printf("\n");
  }
  printf("      +");
  for(thisidx = 0; thisidx <= MAXWORDLEN; thisidx++)
  {
    printf("---");
  }
  printf("\n       ");
  for(thisidx = 0; thisidx < MAXWORDLEN; thisidx++)
  {
    printf("%2d ", thisidx + 1);
  }
  printf(">%d\n", MAXWORDLEN);

  return 0;
}
```

14. *Write a program to print a histogram of the frequencies of different characters in its input.*

```
#include <stdio.h>

/* NUM_CHARS should really be CHAR_MAX but K&R
haven't covered that at this stage in the book */
#define NUM_CHARS 256

int main(void)
{
  int c;
  long freqarr[NUM_CHARS + 1];
```

```c
long thisval = 0;
long maxval = 0;
int thisidx = 0;

for(thisidx = 0; thisidx <= NUM_CHARS; thisidx++)
{
  freqarr[thisidx] = 0;
}

while((c = getchar()) != EOF)
{
  if(c < NUM_CHARS)
  {
    thisval = ++freqarr[c];
    if(thisval > maxval)
    {
      maxval = thisval;
    }
  }
  else
  {
    thisval = ++freqarr[NUM_CHARS];
    if(thisval > maxval)
    {
      maxval = thisval;
    }
  }
}

for(thisval = maxval; thisval > 0; thisval--)
{
  printf("%4d  |", thisval);
  for(thisidx = 0; thisidx <= NUM_CHARS; thisidx++)
  {
    if(freqarr[thisidx] >= thisval)
    {
      printf("*");
    }
    else if(freqarr[thisidx] > 0)
    {
```

```
        printf(" ");
      }
    }
    printf("\n");
  }
  printf("      +");
  for(thisidx = 0; thisidx <= NUM_CHARS; thisidx++)
  {
    if(freqarr[thisidx] > 0)
    {
      printf("-");
    }
  }
  printf("\n       ");
  for(thisidx = 0; thisidx < NUM_CHARS; thisidx++)
  {
    if(freqarr[thisidx] > 0)
    {
      printf("%d", thisidx / 100);
    }
  }
  printf("\n       ");
  for(thisidx = 0; thisidx < NUM_CHARS; thisidx++)
  {
    if(freqarr[thisidx] > 0)
    {
      printf("%d", (thisidx - (100 * (thisidx /
100))) / 10 );
    }
  }
  printf("\n       ");
  for(thisidx = 0; thisidx < NUM_CHARS; thisidx++)
  {
    if(freqarr[thisidx] > 0)
    {
      printf("%d", thisidx - (10 * (thisidx / 10)));
    }
  }
  if(freqarr[NUM_CHARS] > 0)
  {
    printf(">%d\n", NUM_CHARS);
```

```
  }

  printf("\n");

  return 0;
}
```

15. *Rewrite the temperature conversion program of Section 1.2 to use a function for conversion.*

```
#include <stdio.h>

float FtoC(float f)
{
  float c;
  c = (5.0 / 9.0) * (f - 32.0);
  return c;
}

int main(void)
{
  float fahr, celsius;
  int lower, upper, step;

  lower = 0;
  upper = 300;
  step = 20;

  printf("F     C\n\n");
  fahr = lower;
  while(fahr <= upper)
  {
    celsius = FtoC(fahr);
    printf("%3.0f %6.1f\n", fahr, celsius);
    fahr = fahr + step;
  }
  return 0;
}
```

16. *Revise the main routine of the longest-line program so it will correctly print the length of arbitrarily long input lines, and as much as*

*possible of the text.*

```
#include <stdio.h>
#define MAXLENGTH 20
int getline(char [],int);
void copy(char [],char []);
int main()
{
    int len,max=0;
    char line[MAXLENGTH],longest[MAXLENGTH];
    while((len=getline(line,MAXLENGTH))>0)
        if(len>max){
            max=len;
            copy(longest,line);
        }
    if(max>0){
        if(max>MAXLENGTH){
            printf("\n\nStorage limit exceeded
by : %d",max-MAXLENGTH);
            printf("\nString length : %d",max);
            printf("\n%s",longest);
        }
        else
            printf("%s",longest);
    }
    return 0;
}

int getline(char line[],int limit)
{
    int i,c;
    for(i=0;i<limit-1&&(((c=getchar())!=EOF)&&(c!
='\n'));i++)
        line[i]=c;
    if(i==(limit-1)){
        while((c=getchar())!='\n'){
            ++i;
        }
    }
    if(c=='\n'){
        line[i]=c;
        ++i;
```

```
    }
    line[i]='\0';
    return i;
}

void copy(char to[],char from[])
{
    int i=0;
    while((to[i]=from[i])!='\0')
        ++i;
}
```

17. *Write a program to print all input lines that are longer than 80 characters.*

```
#include<stdio.h>

#define MAXLINE 1000
#define MAXLENGTH 81

int getline(char [], int max);
void copy(char from[], char to[]);

int main()
{
  int len = 0; /* current line length */
  char line[MAXLINE]; /* current input line */

  while((len = getline(line, MAXLINE)) > 0)
    {
      if(len > MAXLENGTH)
    printf("LINE-CONTENTS:  %s\n", line);
    }

  return 0;
}



int getline(char line[], int max)
{
```

```
  int i = 0;
  int c = 0;

  for(i = 0; ((c = getchar()) != EOF) && c != '\n' &&
i < max - 1; ++i)
    line[i] = c;

  if(c == '\n')
    line[i++] = c;

  line[i] = '\0';

  return i;
}
```

18. *Write a program to remove all trailing blanks and tabs from each line of input, and to delete entirely blank lines.*

```
#include <stdio.h>
#define MAXLINE 10000

char line[MAXLINE+1];
int getline(void);

int main(void)
{
    extern char line[];
    int len, head, tail, inn;
    while((len=getline()) > 0) {
        for(head = 0; line[head] == ' ' ||
            line[head] == '\t'; head++);
        for(tail = len; line[tail] == ' ' ||
            line[tail] == '\t' ||
            line[tail] == '\n' ||
            line[tail] == '\0';tail--);
        if(tail - head >= 0){
            for(inn = head; inn <= tail; inn++)
                putchar(line[inn]);
            putchar('\n');
            putchar('\0');
        }
    }
```

```
    return 0;
}

int getline(void)
{
    extern char line[];
    int c, i;
    for(i = 0; i < MAXLINE-1 && (c=getchar())
        != EOF && c != '\n'; ++i) line[i] = c;
    if (c == '\n') {
        line[i] = c;
        ++i;
    }
    line[i] = '\0';
    return i;
}
```

19. *Write a function* `reverse(s)` *that reverses the character string* `s` *. Use it to write a program that reverses its input a line at a time.*

```
#include <stdio.h>

#define MAXLINE 1000

/* reverse a line, discard empty lines */

int getline(char s[], int max);
void reverse(char s[]);

int
main(void)
{
    int len, i;
    char line[MAXLINE], longest[MAXLINE];

    while ((len = getline(line, MAXLINE)) != 0) {
        if (len > 1) {
            reverse(line);
            printf("%s\n", line);
        }
    }
```

```
    return 0;
}

int getline(char s[], int max) {
    int i, c;
    for (i=0; i<max-1 && (c=getchar())!=EOF && c!
='\n'; ++i) {
        s[i] = c;
    }
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}

void reverse(char s[]) {
    int i, j;
    char temp;

    for (j = 0; s[j] != '\0'; ++j)
        ;
    --j;

    if (s[j] == '\n') {
        s[j] = '\0';
        --j;
    }

    for (i = 0; i < j; i++) {
        temp = s[i];
        s[i] = s[j];
        s[j] = temp;
        --j;
    }
}
```

20. *Write a program* `detab` *that replaces tabs in the input with the proper number of blanks to space to the next tab stop. Assume a fixed set of tab stops, say every* n *columns. Should* n *be a variable or a symbolic parameter?*

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

#define TAB 7

int main(void) {
  int c,i;

  i = 0;

  while ((c = getchar()) != EOF) {
    i++;
    if (c == '\n')
      i = 0; /* reset column counter */
    if (c == '\t') {
      while ((i % TAB) != 0) {
        putchar(' ');
        i++;
      }
    } else {
      putchar(c);
    }

  }
  return(0);
}
```

21. *Write a program* `entab` *that replaces strings of blanks with the minimum number of tabs and blanks to achieve the same spacing. Use the same stops as for* `detab`*. When either a tab or a single blank would suffice to reach a tab stop, which should be given preference?*

```c
#include <stdio.h>

#define TAB 5 // Number of spaces of one tab.

main()
{
  int c, i = 0, j;
```

```
  while((c = getchar()) != EOF){

    if(c == ' '){

      ++i; // This is a counter of white spaces.

      if((i % TAB) == 0) // Every group of a number
of 'TAB'
        putchar('\t');   // spaces is replaced by a
tab.
    }

    else{

      for(j = 0; j < (i % TAB); ++j)   // Every group
smaller than
        putchar(' ');             // 'TAB' spaces is
untouched.

      putchar(c); // Well, there exist other
characters but spaces.

      if(i != 0) // Once some text is found, the
counter is reset.
        i = 0;
    }
  }
}
```
22. *Write a program to "fold" long input lines into two or more shorter lines after the last non-blank character that occurs before the* `n`*-th column of input. Make sure your program does something intelligent with very long lines, and if there are no blanks or tabs before the specified column.*

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

#define MAXLINE 10000
#define LIMIT 20

int getline(char s[], int lim);
```

```c
int cut(char s[], int start);

int main(void)
{
    int i;
    int length;
    char string[MAXLINE];

    while ((length = getline(string,MAXLINE)) > 0) {
        if (length > LIMIT) {
            cut(string, 0);
            printf("%s",string);
        }
        else
            printf("%s",string);
    }
    return 0;
}

int cut(char s[], int start)
{
    int i,j;
    int spaces = 0;

    for(i=start+LIMIT; i > start && s[i] != ' ' &&
s[i] != '\t'; --i);
    if (i == start) {
        for(i=start; s[i] != ' ' && s[i] != '\t'; +
+i);
        s[i] = '\n';
    }
    if ((strlen(s) - i) <= LIMIT) {
        s[i] = '\n';
    }
    else if (i > start && i != start) {
        s[i] = '\n';
        cut(s, i);
    }
    return 0;
}
```

```c
int getline(char s[], int lim)
{
    int c, i;

    for (i=0; i<lim-1 && (c=getchar())!=EOF && c!
='\n'; ++i)
        s[i] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}
```

23. *Write a program to remove all comments from a C program. Don't forget to handle quoted strings and character constants properly. C comments do not nest.*

```c
#include <stdio.h>

/* remove comments from C sources */

#define YES 1
#define NO  !YES

int main()
{
    /* c is the current character, c_prev is the
previous one and c_pprev the one before c_prev */
    int c, c_prev='\0', c_pprev = '\0', is_comment =
NO, is_string = NO, closing_symbol;

    while ((c = getchar()) != EOF)
    {
        if (!is_comment)
        {
            /* fix the slash if it is not a comment
*/
            if (!is_string && c_prev == '/' && c !=
'*' && c_pprev != '*')
                putchar('/');
            /* print the char if it is not the
```

```
begining of a comment */
                if (is_string || (c != '/' && (c != '*'
|| c_prev != '/')))
                    putchar(c);
            }
            /* closing the comment */
            if (is_comment && c == '/' && c_prev == '*')
                is_comment = NO;
            /* begining the comment */
            else if (!is_comment && !is_string && c ==
'*' && c_prev == '/')
                is_comment = YES;
            /* closing the string or character, handles
escape sequences \' and \\' */
            else if (is_string && c == closing_symbol &&
(c_prev != '\\' || c_pprev == '\\'))
                is_string = NO;
            /* begining the string or character */
            else if (!is_string && !is_comment && (c ==
'"' || c == '\''))
            {
                is_string = YES;
                closing_symbol = c;
            }
            c_pprev = c_prev;
            c_prev = c;
        }

    return 0;
}
```

24. *Write a program to check a C program for rudimentary syntax errors like unbalanced parentheses, brackets and braces. Don't forget about quotes, both single and double, escape sequences, and comments. (This program is hard if you do it in full generality.)*

```
#include <stdio.h>

#define MAX_INPUT_LENGTH        10000

#define NORMAL          0
```

```c
#define SINGLE_QUOTE  1
#define DOUBLE_QUOTE  2
#define SLASH         3
#define MULTI_COMMENT 4
#define INLINE_COMMENT    5
#define STAR      6

int state_from_normal(char symbol, char prev_symbol)
{
    int state = NORMAL;

    if (symbol == '\'' && prev_symbol != '\\')
        state = SINGLE_QUOTE;
    else if (symbol == '"')
        state = DOUBLE_QUOTE;
    else if (symbol == '/')
        state = SLASH;

    return state;
}

int state_from_single_quote(char symbol, char
prev_symbol, char pre_prev_symbol)
{
    int state = SINGLE_QUOTE;

    if (symbol == '\'' && (prev_symbol != '\\' ||
pre_prev_symbol == '\\'))
        state = NORMAL;

    return state;
}

int state_from_double_quote(char symbol, char
prev_symbol, char pre_prev_symbol)
{
    int state = DOUBLE_QUOTE;

    if (symbol == '"' && (prev_symbol != '\\' ||
pre_prev_symbol == '\\'))
        state = NORMAL;
```

```
    return state;
}

int state_from_slash(char symbol)
{
    int state = SLASH;

    if (symbol == '*')
        state = MULTI_COMMENT;
    else if (symbol == '/')
        state = INLINE_COMMENT;
    else
        state = NORMAL;

    return state;
}

int state_from_multi_comment(char symbol)
{
    int state = MULTI_COMMENT;

    if (symbol == '*')
        state = STAR;

    return state;
}

int state_from_star(char symbol)
{
    int state = STAR;

    if (symbol == '/')
        state = NORMAL;
    else if (symbol != '*')
        state = MULTI_COMMENT;

    return state;
}

int state_from_inline_comment(char symbol)
```

```
{
    int state = INLINE_COMMENT;

    if (symbol == '\n')
        state = NORMAL;

    return state;
}

int state_from(int prev_state, char symbol, char
prev_symbol, char pre_prev_symbol)
{
    if (prev_state == NORMAL)
        return state_from_normal(symbol,
prev_symbol);
    else if (prev_state == SINGLE_QUOTE)
        return state_from_single_quote(symbol,
prev_symbol, pre_prev_symbol);
    else if (prev_state == DOUBLE_QUOTE)
        return state_from_double_quote(symbol,
prev_symbol, pre_prev_symbol);
    else if (prev_state == SLASH)
        return state_from_slash(symbol);
    else if (prev_state == MULTI_COMMENT)
        return state_from_multi_comment(symbol);
    else if (prev_state == INLINE_COMMENT)
        return state_from_inline_comment(symbol);
    else if (prev_state == STAR)
        return state_from_star(symbol);
    else
        return -1;
}

char opening_symbol(char symbol)
{
    if (symbol == ')')
        return '(';
    else if (symbol == ']')
        return '[';
    else if (symbol == '}')
        return '{';
```

```
    else
        return '\0';
}

int is_valid_closing(char symbol, char nests[], int
nest_index)
{
    return nest_index > 0 && nests[nest_index-1] ==
opening_symbol(symbol);
}

int main(void)
{
    char nests[MAX_INPUT_LENGTH] = { '\0' };
    int  nest_index = 0;

    char input;
    char symbol = '\0';
    char prev_symbol = '\0';
    char pre_prev_symbol;

    int state = NORMAL;
    int prev_state;

    int line = 1, column = 0;

    while ((input = getchar()) != EOF) {
        column++;

        pre_prev_symbol = prev_symbol;
        prev_symbol     = symbol;
        symbol          = input;

        prev_state = state;
        state = state_from(prev_state, symbol,
prev_symbol, pre_prev_symbol);

        if (symbol == '\n') {
            line++;
            column = 0;
        } else if (state == NORMAL) {
```

```
            if (symbol == '(' || symbol == '[' ||
symbol == '{') {
                nests[nest_index++] = symbol;
            }
            if (symbol == ')' || symbol == ']' ||
symbol == '}') {
                if (is_valid_closing(symbol, nests,
nest_index)) {
                    nests[--nest_index] = '\0';
                } else {
                    printf("Unexpected '%c' at
line %d, column %d\n", symbol, line, column);
                    return 1;
                }
            }
        }
    }

    if (nest_index > 0) {
        printf("Unbalanced '%c'", nests[0]);
        for (int i = 1; i < nest_index; i++) {
            printf(", '%c'", nests[i]);
        }
        printf("\n");
        return 1;
    } else {
        printf("Balanced\n");
    }
}
```

Chapter 2.

1. *Write a program to determine the ranges of* `char` , `short` , `int` , *and* `long` *variables, both* `signed` *and* `unsigned` *, by printing appropriate values from standard headers and by direct computation. Harder if you compute them: determine the ranges of the various floating-point types.*

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

```c
int main(void)
{
  printf("\nBits of type char: %d\n\n",
CHAR_BIT);                              /* IV */

  printf("Maximum numeric value of type char: %d\n",
CHAR_MAX);          /* IV */
  printf("Minimum numeric value of type char: %d\n
\n", CHAR_MIN);       /* IV */

  printf("Maximum value of type signed char: %d\n",
SCHAR_MAX);          /* IV */
  printf("Minimum value of type signed char: %d\n\n",
SCHAR_MIN);        /* IV */

  printf("Maximum value of type unsigned char: %u\n
\n", (unsigned) UCHAR_MAX);     /* SF */  /* IV */

  printf("Maximum value of type short: %d\n",
SHRT_MAX);                  /* IV */
  printf("Minimum value of type short: %d\n\n",
SHRT_MIN);                /* IV */

  printf("Maximum value of type unsigned short: %u\n
\n", (unsigned) USHRT_MAX);    /* SF */  /* IV */


  printf("Maximum value of type int: %d\n",
INT_MAX);                      /* IV */
  printf("Minimum value of type int: %d\n\n",
INT_MIN);                    /* IV */

  printf("Maximum value of type unsigned int: %u\n
\n", UINT_MAX);       /* RB */   /* IV */

  printf("Maximum value of type long: %ld\n",
LONG_MAX);                  /* RB */   /* IV */
  printf("Minimum value of type long: %ld\n\n",
LONG_MIN);                /* RB */   /* IV */
```

```
  printf("Maximum value of type unsigned long: %lu\n
\n", ULONG_MAX);    /* RB */    /* IV */


  return 0;
}
```

2. *Exercise 2-2 discusses a* `for` *loop from the text. Here it is:*

```
  for(i=0; i<lim-1 && (c=getchar()) != '\n' && c !=
EOF; ++i)
    s[i] = c;
```

*Write a loop equivalent to the* `for` *loop above without using* `&&` *or* `||` *.*

```
#include <stdio.h>

#define lim 80

int main()
{
        int i, c;
        char s[lim];

        /* There is a sequence point after the first
operand of ?: */

        for(i=0; i<lim-1 ? (c=getchar()) != '\n' ?
c != EOF : 0 : 0 ; ++i)
                s[i] = c;

        return s[i] ^= s[i]; /* null terminate and
return. */
}
```

2nd for while

```
         i=0;
         while(i<lim-1) {
              if((c=getchar()) != '\n') {
                   if(c != EOF) {
                         s[i] = c;
```

```
                }
            }
            i++;
        }
```

3. *Write the function* `htoi(s)` *, which converts a string of hexadecimal digits (including an optional* `0x` *or* `0X)` *into its equivalent integer value. The allowable digits are* `0` *through* `9,` `a` *through* `f,` *and* `A` *through* `F` *.*

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

unsigned long htoi(const char s[]);

int main(void)
{
        printf("%ld\n", htoi("0xFA9C"));
        printf("%ld\n", htoi("0xFFFF"));
        printf("%ld\n", htoi("0x1111"));
        printf("%ld\n", htoi("0xBCDA"));
        return 0;
}

unsigned long htoi(const char s[])
{
        unsigned long n = 0;

        for (int i = 0; s[i] != '\0'; i++) {
                int c = tolower(s[i]);
                if (c == '0' && tolower(s[i+1]) ==
'x')
                        i++;
                else if (c >= '0' && c <= '9')
                        n = 16 * n + (c - '0');
                else if (c >= 'a' && c <= 'f')
                        n = 16 * n + (c - 'a' + 10);
        }
        return n;
}
```

4. *Write an alternate version of* `squeeze(s1,s2)` *that deletes each*

*character in the string `s1` that matches any character in the string `s2` .*

```
void squeeze2(char s[], char t[]) {
    int i, j, k;
    for (k = 0; t[k] != '\0'; k++) {
        for (i = j = 0; s[i] != '\0'; i++)
            if (s[i] != t[k])
                s[j++] = s[i];
        s[j] = '\0';
    }
}
```

5. *Write the function `any(s1,s2)` , which returns the first location in the string `s1` where any character from the string `s2` occurs, or `-1` if `s1` contains no characters from `s2` . (The standard library function `strpbrk` does the same job but returns a pointer to the location.)*

```
#include <stdio.h>

int any(char s1[],char s2[]);

int main() {
  char s1[] = "hello world";
  char s2[] = "pjwyh";
  printf("%d\n", any(s1,s2));
  return 0;
}

int
any(char s1[], char s2[]) {
  int i, j;
  int ret = -1;
  for(j=0; s2[j] != '\0'; j++)
    for(i=0; s1[i] != '\0'; i++)
      if(s1[i] == s2[j])
        if(ret<0)
          ret = i;
        else if(i<ret)
          ret = i;
  return ret;
}
```

6. *Write a function* `setbits(x,p,n,y)` *that returns* `x` *with the* `n` *bits that begin at position* `p` *set to the rightmost* `n` *bits of* `y`, *leaving the other bits unchanged.*

```
#include <stdio.h>

unsigned setbits(unsigned x, int p, int n, unsigned
y)
{
  return (x & ((~0 << (p + 1)) | (~(~0 << (p + 1 -
n))))) | ((y & ~(~0 << n)) << (p + 1 - n));
}

int main(void)
{
  unsigned i;
  unsigned j;
  unsigned k;
  int p;
  int n;

  for(i = 0; i < 30000; i += 511)
  {
    for(j = 0; j < 1000; j += 37)
    {
      for(p = 0; p < 16; p++)
      {
        for(n = 1; n <= p + 1; n++)
        {
          k = setbits(i, p, n, j);
          printf("setbits(%u, %d, %d, %u) = %u\n", i,
p, n, j, k);
        }
      }
    }
  }
  return 0;
}
```

7. *Write a function* `invert(x,p,n)` *that returns* `x` *with the* `n` *bits that begin at position* `p` *inverted (i.e., 1 changed into 0 and vice versa), leaving the others unchanged.*

```c
unsigned invert(unsigned x, int p, int n)
{
    return x ^ (~(~0U << n) << p);
}

/*
main driver added, in a hurry while tired, by RJH.
Better test driver suggestions are welcomed!

main driver fixed by Flash Gordon as it was passing
the parameters in the wrong order and
hex is a more useful output format than decimal for
checking the result. Also start at 0
for n,p as they are valid inputs.
*/

#include <stdio.h>

int main(void)
{
  unsigned x;
  int p, n;

  for(x = 0; x < 700; x += 49)
    for(n = 0; n < 8; n++)
      for(p = 0; p < 8; p++)
        printf("%x, %d, %d: %x\n", x, p, n, invert(x,
p, n));
  return 0;
}
```

8. *Write a function `rightrot(x,n)` that returns the value of the integer `x` rotated to the right by `n` bit positions.*

```c
unsigned rightrot(unsigned x, unsigned n)
{
    while (n > 0) {
        if ((x & 1) == 1)
            x = (x >> 1) | ~(~0U >> 1);
        else
```

```
            x = (x >> 1);
        n--;
    }
    return x;
}

/* main driver added, in a hurry while tired, by RJH.
Better test driver suggestions are welcomed! */

#include <stdio.h>

int main(void)
{
  unsigned x;
  int n;

  for(x = 0; x < 700; x += 49)
    for(n = 1; n < 8; n++)
      printf("%u, %d: %u\n", x, n, rightrot(x, n));
  return 0;
}
```

9. *n a two's complement number system,* `x &= (x-1)` *deletes the rightmost 1-bit in* `x` *. Explain why. Use this observation to write a faster version of* `bitcount` *.*

```
/* bitcount:  count 1 bits in x */
int bitcount(unsigned x)
{
    int b;

    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            b++;
    return b;
}
```

10. *Rewrite the function lower, which converts upper case letters to lower case, with a conditional expression instead of* `if-else` *.*

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

```
unsigned char llower(char);

int main(void)
{
    int i;
    char test[] = "AaBbCcdDeE1234ZzyY";
    i = 0;
    puts(test);
    while(test[i] != '\0')putchar(llower(test[i++]));
    putchar('\n');
}

unsigned char llower(char x)
{
    return (x >= 'A' && x <= 'Z') ? x = x - 'A'
+'a' : x;
}
```

chapter 3.

1. *Our binary search makes two tests inside the loop, when one would suffice (at the price of more tests outside). Write a version with only one test inside the loop and measure the difference in run-time.*

```
#include <stdio.h>

/* find x in v[] */
int binsearch(int x, int v[], int n);

/*
  The main is here for the purpose of a built in test

 */


int main(void)
{
  int test[]={1,3,5,7,9,11,13};
```

```
  int i;
  for(i=(sizeof(test)/sizeof(int))-1; i>=0; --i)
    printf("looking for %d. Index=%d
\n",test[i],binsearch(test[i], test, sizeof(test)/
sizeof(*test)));

  return 0;

}
/* n = size of array v */

int binsearch(int x, int v[], int n)
{
  int low, high, mid;

  low = 0;
  high = n-1;

  while(low < high) {
    mid = (low+high)/2;
    if(x <= v[mid])
      high=mid;

    else
      low = mid+1;
  }

  return (x == v[low])?low : -1;

}
```

2. *Write a function* `escape(s,t)` *that converts characters like newline and tab into visible escape sequences like* `\n` *and* `\t` *as it copies the string* `t` *to* `s` *. Use a* `switch` *. Write a function for the other direction as well, converting escape sequences into the real characters.*

```
#include <stdio.h>

void escape(char * s, char * t);
```

```
void unescape(char * s, char * t);

int main(void) {
    char text1[50] = "\aHello,\n\tWorld! Mistakee\b
was \"Extra 'e'\"!\n";
    char text2[51];

    printf("Original string:\n%s\n", text1);

    escape(text2, text1);
    printf("Escaped string:\n%s\n", text2);

    unescape(text1, text2);
    printf("Unescaped string:\n%s\n", text1);

    return 0;
}


/*  Copies string t to string s, converting special
    characters into their appropriate escape
sequences.
    The "complete set of escape sequences" found in
    K&R Chapter 2 is used, with the exception of:

    \? \' \ooo \xhh

    as these can be typed directly into the source
code,
    (i.e. without using the escape sequences
themselves)
    and translating them is therefore ambiguous.
*/

void escape(char * s, char * t) {
    int i, j;
    i = j = 0;

    while ( t[i] ) {

        /*  Translate the special character, if we
```

```
have one  */
        switch( t[i] ) {
        case '\n':
            s[j++] = '\\';
            s[j] = 'n';
            break;

        case '\t':
            s[j++] = '\\';
            s[j] = 't';
            break;

        case '\a':
            s[j++] = '\\';
            s[j] = 'a';
            break;

        case '\b':
            s[j++] = '\\';
            s[j] = 'b';
            break;

        case '\f':
            s[j++] = '\\';
            s[j] = 'f';
            break;

        case '\r':
            s[j++] = '\\';
            s[j] = 'r';
            break;

        case '\v':
            s[j++] = '\\';
            s[j] = 'v';
            break;

        case '\\':
            s[j++] = '\\';
            s[j] = '\\';
```

```
            break;

        case '\"':
            s[j++] = '\\';
            s[j] = '\"';
            break;

        default:

            /*  This is not a special character, so 
just copy it  */

            s[j] = t[i];
            break;
        }
        ++i;
        ++j;
    }
    s[j] = t[i];    /*  Don't forget the null 
character  */
}


/*  Copies string t to string s, converting escape 
sequences
    into their appropriate special characters. See 
the comment
    for escape() for remarks regarding which escape 
sequences
    are translated.
*/

void unescape(char * s, char * t) {
    int i, j;
    i = j = 0;

    while ( t[i] ) {
        switch ( t[i] ) {
        case '\\':

            /*  We've found an escape sequence, so 
```

```
translate it  */
            switch( t[++i] ) {
            case 'n':
                s[j] = '\n';
                break;

            case 't':
                s[j] = '\t';
                break;

            case 'a':
                s[j] = '\a';
                break;

            case 'b':
                s[j] = '\b';
                break;

            case 'f':
                s[j] = '\f';
                break;

            case 'r':
                s[j] = '\r';
                break;

            case 'v':
                s[j] = '\v';
                break;

            case '\\':
                s[j] = '\\';
                break;

            case '\"':
                s[j] = '\"';
                break;

            default:
```

```
                /*  We don't translate this escape
                    sequence, so just copy it
verbatim  */

                s[j++] = '\\';
                s[j] = t[i];
            }
            break;

        default:

            /*  Not an escape sequence, so just copy
the character  */

            s[j] = t[i];
        }
        ++i;
        ++j;
    }
    s[j] = t[i];    /*  Don't forget the null
character  */
}
```

3. *Write a function `expand(s1,s2)` that expands shorthand notations like `a-z` in the string `s1` into the equivalent complete list `abc...xyz` in `s2`. Allow for letters of either case and digits, and be prepared to handle cases like `a-b-c` and `a-z0-9` and `-a-z`. Arrange that a leading or trailing `-` is taken literally.*

```
#include <stdio.h>


void expand(char s1[], char s2[]) {

    char c, d, e;
    int i, j;
    i = j = 0;

    while ('\0' != (c = s1[i++])) {
        if (' ' != c && '-' == s1[i] && '\0' != s1[i
```

```
+ 1]) {
            i++;
            d = s1[i];
            if (d < c) {
                while (c > d) {
                    s2[j++] = c--;
                }
            }
            else {
                while (c < d) {
                    s2[j++] = c++;
                }
            }
        }
        else {
            s2[j++] = c;
        }
    }
    s2[j] = '\0';
}


main() {


    char s1[512] = "-a-z 0-9 a-d-f -0-2 some text 1-1
WITH CAPITALS! 0-0 5-3 -";
    char s2[512];
    expand(s1, s2);
    printf("%s\n", s2);

}
```

4.*n a two's complement number representation, our version of* `itoa` *does not handle the largest negative number, that is, the value of* `n` *equal to* `-(2` *to the power* `(wordsize - 1))` *. Explain why not. Modify it to print that value correctly regardless of the machine on which it runs.*

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

```c
#include <limits.h>

void reverse(char s[]) {
    int length = strlen(s);
    int c, i, j;

    for (i = 0, j = length - 1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}

void itoa(int n, char s[]) {

    int i, sign;
    unsigned int n2;
    i = 0;


    if ((sign = n) < 0) {
        n2 = -n;
    }
    else {
        n2 = n;
    }

    do {
        s[i++] = (n2 % 10) + '0';
    }
    while ((n2 /= 10) > 0);
    if (sign < 0) {
        s[i++] = '-';
    }
    s[i] = '\0';

    reverse(s);


}
```

```
main() {

    char s[128];
    itoa(INT_MIN, s);
    printf("%d is converted to %s.\n", INT_MIN, s);

}
```

5. *Write the function `itob(n,s,b)` that converts the integer `n` into a base `b` character representation in the string `s`. In particular, `itob(n,s,16)` formats `n` as a hexadecimal integer in `s`.*

```
#include <stdio.h>

void itob(int n, char s[], int b);
void reverse(char s[]);

int main(void)
{
    int n;
    char s[100];

    for (int i = 2; i <= 20; i++) {
        itob(255, s, i);
        printf("decimal 255 in base %-2d : %s\n", i,
s);
    }

    return 0;
}

void itob(int n, char s[], int b)
{
    int i, sign, r;

    sign = n;
    i = 0;
    do {
        r = n % b;
        if (sign < 0)
```

```c
            r = -r;
        s[i++] = (r > 9 ? (r-10 + 'A') : (r + '0'));
    } while (n /= b);

    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';

    reverse(s);
}

void reverse(char s[])
{
    int i, j, t;

    for (j = 0; s[j] != '\0'; j++)
        ;
    for (i = 0, --j; j > i; i++, j--) {
        t = s[j];
        s[j] = s[i];
        s[i] = t;
    }
}
```

6. *Write a version of* `itoa` *that accepts three arguments instead of two. The third argument is a minimum field width; the converted number must be padded with blanks on the left if necessary to make it wide enough.*

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

void itoa(int n, char s[], int width);
void reverse(char s[]);

int main(void) {
    char buffer[20];

    itoa(INT_MIN, buffer, 7);
    printf("Buffer:%s\n", buffer);

    return 0;
}
```

```c
void itoa(int n, char s[], int width) {
    int i, sign;

    if ((sign = n) < 0)
        n = -n;
    i = 0;
    do {
        s[i++] = n % 10 + '0';
        printf("%d %% %d + '0' = %d\n", n, 10,
s[i-1]);
    } while ((n /= 10) > 0);
    if (sign < 0)
        s[i++] = '-';

    while (i < width )     /*  Only addition to
original function  */
        s[i++] = ' ';

    s[i] = '\0';
    reverse(s);
}

void reverse(char s[]) {
    int c, i, j;
    for ( i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

**chapter 4**

**1.** *Write the function* `strrindex(s,t)` *, which returns the position of the rightmost occurrence of* `t` *in* `s` *, or -1 if there is none.*

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

int strrindex(char *s, char *t)
```

```
{
    int i, j, flag, slen = strlen(s), tlen =
strlen(t);
    for(i = slen - tlen; i >= 0; i--)
    {
        flag = 1;
        for(j = i; j < i + tlen; j++)
        {
            if(s[j] != t[j-i])
            {
                flag = 0;
                break;
            }
        }
        if(flag == 1)
            return i;
    }
    return -1;
}

int main()
{
    char s[] = "When I get older, I will be
stronger";
    char t[] = "I";
    printf("%d\n", strrindex(s, t));
    return 0;
}
```

**2.** *Extend `atof` to handle scientific notation of the form `123.45e-6` where a floating-point number may be followed by e or E and an optionally signed exponent.*

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

double atof(char s[]);

int main(void)
{
    printf("%f\n", atof("123.45e-6"));
```

```
}

double atof(char s[])
{
    double val, power, base, p;
    int i, sign, exp;

    for (i = 0; isspace(s[i]); i++)
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '-' || s[i] == '+')
        ++i;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10.0;
    }
    if (s[i] == 'e' || s[i] == 'E')
        i++;
    else
        return sign * val/power;
    base = (s[i] == '-') ? 0.1 : 10.0; /* 10^(-n) =
1/10^n = (1/10)^n = (0.1)^n */
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (exp = 0; isdigit(s[i]); i++)
        exp = 10 * exp + (s[i] - '0');
    for (p = 1; exp > 0; --exp)
        p = p * base;

    return (sign * (val/power)) * p;
}
```

3.*Given the basic framework, it's straightforward to extend the calculator. Add the modulus (* `%` *) operator and provisions for negative numbers.*

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

#define MAXOP 100
#define NUMBER 0
#define MAXVAL 100
#define BUFSIZE 100

int getop(char *);
void push(double);
double pop(void);
int getch(void);
void ungetch(int);
void viewstack(void);

int bufp = 0;
char buf[BUFSIZE];
int sp = 0;
double val[MAXVAL];


void push(double f)
{
    if(sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n",
f);
}

double pop(void)
{
    if(sp > 0)
        return val[--sp];
    else
    {
        printf("error: stack empty\n");
        exit(1);
```

```c
        //return 0.0;
    }
}

int getch(void)
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)
{
    if(bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}

int getop(char *s)
{
    int i, c, d;
    while((s[0] = c = getch()) == ' ' || c == '\t');
    s[1] = '\0';
    if(!isdigit(c) && c != '.' && c != '-')
        return c;
    if(c == '-')
    {
        d = getch();
        if(d == ' ')
            return c;
        else
            ungetch(d);
    }
    i = 0;
    if(isdigit(c) || c == '-')
        while(isdigit(s[++i] = c = getch()));
    if(c == '.')
        while(isdigit(s[++i] = c = getch()));
    s[i] = '\0';
    if(c != EOF)
        ungetch(c);
    return NUMBER;
```

```
}

void viewstack(void)
{
    int i;
    printf("\nstack:\n");
    for(i = sp - 1; i >= 0; i--)
        printf("%lf\n", val[i]);
}

int main()
{
    int type;
    double op2;
    char s[MAXOP];

    while((type = getop(s)) != EOF)
    {
        //viewstack();  Use this function if you wish
to see the stack after every iteration
        switch(type)
        {
            case NUMBER:
                push(atof(s));
                break;
            case '+':
                push(pop() + pop());
                break;
            case '*':
                push(pop() * pop());
                break;
            case '-':
                op2 = pop();
                push(pop() - op2);
                break;
            case '/':
                op2 = pop();
                if(op2 != 0.0)
                    push(pop() / op2);
                else
                    printf("error: zero divisor\n");
```

```
                break;
            case '%':
                op2 = pop();
                if(op2 != 0.0)
                    push(fmod(pop(), op2));
                else
                    printf("error: division by zero
\n");
                break;
            case '\n':
                printf("\t%.8g\n", pop());
                break;
            default:
                printf("error: unknown command %s
\n", s);
                break;
        }
    }
    return 0;
}
```

4. *Add commands to print the top element of the stack without popping, to duplicate it, and to swap the top two elements. Add a command to clear the stack.*

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

#define MAXOP 100
#define NUMBER 0
#define MAXVAL 100
#define BUFSIZE 100

int getop(char *);
void push(double);
double pop(void);
```

```c
int getch(void);
void ungetch(int);
void viewstack(void);
void showTop(void);
void swap(void);
void duplicate(void);
void clearStack(void);

int bufp = 0;
char buf[BUFSIZE];
int sp = 0;
double val[MAXVAL];


void push(double f)
{
    if(sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n",
f);
}

double pop(void)
{
    if(sp > 0)
        return val[--sp];
    else
    {
        printf("error: stack empty\n");
        exit(1);
        //return 0.0;
    }
}

int getch(void)
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)
```

```c
{
    if(bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}

int getop(char *s)
{
    int i, c, d;
    while((s[0] = c = getch()) == ' ' || c == '\t');
    s[1] = '\0';
    if(!isdigit(c) && c != '.' && c != '-')
        return c;
    if(c == '-')
    {
        d = getch();
        if(d == ' ')
            return c;
        else
            ungetch(d);
    }
    i = 0;
    if(isdigit(c) || c == '-')
        while(isdigit(s[++i] = c = getch()));
    if(c == '.')
        while(isdigit(s[++i] = c = getch()));
    s[i] = '\0';
    if(c != EOF)
        ungetch(c);
    return NUMBER;
}

void viewstack(void)
{
    int i;
    printf("\nstack:\n");
    for(i = sp - 1; i >= 0; i--)
        printf("%lf\n", val[i]);
}
```

```
void showTop(void)
{
    sp > 0 ? printf("\t%.8g\n", val[sp-1]) :
printf("stack is empty\n");
}

void swap(void)
{
    double temp;
    if(sp < 1)
        printf("error: stack has less than 2
elements, can't swap\n");
    else
    {
        temp = val[sp - 1];
        val[sp - 1] = val[sp - 2];
        val[sp - 2] = temp;
    }
}

void duplicate(void)
{
    if(sp > MAXVAL - 1)
        printf("error: stack is full, can't duplicate
\n");
    else
    {
        double temp = pop();
        push(temp);
        push(temp);
        ++sp;
    }
}

void clearStack(void)
{
    sp = 0;
}

int main()
{
```

```
    int type;
    double op2;
    char s[MAXOP];

    while((type = getop(s)) != EOF)
    {
        //viewstack();  Use this function if you wish
to see the stack after every iteration
        switch(type)
        {
            case NUMBER:
                push(atof(s));
                break;
            case '+':
                push(pop() + pop());
                break;
            case '*':
                push(pop() * pop());
                break;
            case '-':
                op2 = pop();
                push(pop() - op2);
                break;
            case '/':
                op2 = pop();
                if(op2 != 0.0)
                    push(pop() / op2);
                else
                    printf("error: zero divisor\n");
                break;
            case '%':
                op2 = pop();
                if(op2 != 0.0)
                    push(fmod(pop(), op2));
                else
                    printf("error: division by zero
\n");
                break;
            case '?': // show top item on stack
without popping
                showTop();
```

```
                break;
            case '~': // swap top two elements of
the stack
                swap();
                break;
            case '#': // duplicate the top element
                duplicate();
                break;
            case '!': // clearStack
                clearStack();
                break;
            case '\n':
                printf("\t%.8g\n", pop());
                break;
            default:
                printf("error: unknown command %s
\n", s);
                break;
        }
    }
    return 0;
}
```

5. *Add access to library functions like `sin`, `exp`, and `pow`. See `<math.h>` in Appendix B, Section 4.*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>

#define MAXOP 100
#define NUMBER 0
#define MAXVAL 100
#define BUFSIZE 100
#define IDENTIFIER 1

int getop(char *);
void push(double);
```

```c
double pop(void);
int getch(void);
void ungetch(int);
void viewstack(void);
void showTop(void);
void swap(void);
void duplicate(void);
void clearStack(void);
void mathfunc(char *);

int bufp = 0;
char buf[BUFSIZE];
int sp = 0;
double val[MAXVAL];


void push(double f)
{
    if(sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n",
f);
}

double pop(void)
{
    if(sp > 0)
        return val[--sp];
    else
    {
        printf("error: stack empty\n");
        exit(1);
        //return 0.0;
    }
}

int getch(void)
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}
```

```
void ungetch(int c)
{
    if(bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}

int getop(char *s)
{
    int i, c, d, flag;
    while((s[0] = c = getch()) == ' ' || c == '\t');
    s[1] = '\0';
    if(!isalnum(c) && c != '.' && c != '-')
        return c;
    if(c == '-')
    {
        d = getch();
        if(d == ' ')
            return c;
        else
            ungetch(d);
    }
    i = 0;
    if(isalnum(c) || c == '-')
        while(isalnum(s[++i] = c = getch()));
    if(c == '.')
        while(isalnum(s[++i] = c = getch()));
    s[i] = '\0';
    if(c != EOF)
        ungetch(c);
    flag = 1;
    for(i = 0; s[i] != '\0'; i++)
        if(!isalpha(s[i]))
        {
            flag = 0;
            break;
        }
    if(flag == 1)
        return IDENTIFIER;
```

```
    else
        return NUMBER;
}

void viewstack(void)
{
    int i;
    printf("\nstack:\n");
    for(i = sp - 1; i >= 0; i--)
        printf("%lf\n", val[i]);
}

void showTop(void)
{
    sp > 0 ? printf("\t%.8g\n", val[sp-1]) :
printf("stack is empty\n");
}

void swap(void)
{
    double temp;
    if(sp < 1)
        printf("error: stack has less than 2
elements, can't swap\n");
    else
    {
        temp = val[sp - 1];
        val[sp - 1] = val[sp - 2];
        val[sp - 2] = temp;
    }
}

void duplicate(void)
{
    if(sp > MAXVAL - 1)
        printf("error: stack is full, can't duplicate
\n");
    else
    {
        double temp = pop();
        push(temp);
```

```
        push(temp);
        ++sp;
    }
}

void clearStack(void)
{
    sp = 0;
}

void mathfunc(char *s)
{
    if(strcmp(s, "sin") == 0)
    {
        if(sp < 1)
            printf("error: stack is empty, can't use 
sin function\n");
        else
            push(sin(pop()));
    }
    else if(strcmp(s, "cos") == 0)
    {
        if(sp < 1)
            printf("error: stack is empty, can't use 
cos function\n");
        else
            push(cos(pop()));
    }
    else if(strcmp(s, "exp") == 0)
    {
        if(sp < 1)
            printf("error: stack is empty, can't use 
exp function\n");
        else
            push(exp(pop()));
    }
    else if(strcmp(s, "pow") == 0)
    {
        if(sp < 2)
            printf("error: stack has less than 2 
elements, can't use pow function\n");
```

```
        else
        {
            double op2;
            op2 = pop();
            push(pow(pop(), op2));
        }
    }
    else
        printf("%s is not a supported function\n",
s);
}

int main()
{
    int type;
    double op2;
    char s[MAXOP];

    while((type = getop(s)) != EOF)
    {
        //viewstack(); // Use this function if you
wish to see the stack after every iteration
        switch(type)
        {
            case NUMBER:
                push(atof(s));
                break;
            case IDENTIFIER:
                mathfunc(s);
                break;
            case '+':
                push(pop() + pop());
                break;
            case '*':
                push(pop() * pop());
                break;
            case '-':
                op2 = pop();
                push(pop() - op2);
                break;
            case '/':
```

```
                op2 = pop();
                if(op2 != 0.0)
                    push(pop() / op2);
                else
                    printf("error: zero divisor\n");
                break;
            case '%':
                op2 = pop();
                if(op2 != 0.0)
                    push(fmod(pop(), op2));
                else
                    printf("error: division by zero
\n");
                break;
            case '?': // show top item on stack
without popping
                showTop();
                break;
            case '~': // swap top two elements of
the stack
                swap();
                break;
            case '#': // duplicate the top element
                duplicate();
                break;
            case '!': // clearStack
                clearStack();
                break;
            case '\n':
                printf("\t%.8g\n", pop());
                break;
            default:
                printf("error: unknown command %s
\n", s);
                break;
        }
        //viewstack();
    }
    return 0;
}
```

6. *Add commands for handling variables. (It's easy to provide twenty-six variables with single-letter names.) Add a variable for the most recently printed value.*

```
#include <stdio.h>
#include <stdlib.h>          /* For atof() */
#include <math.h>
#include <ctype.h>

#define MAXOP    100          /* Max size of operand or
operator. */
#define NUMBER   '0'          /* Signal that a number
was found. */
#define VARIABLE '1'
#define VARMAX   27

int is_first_input = 0;     /* Prevents the solution
from being printed on first
                              input */
double var_array[VARMAX];   /* Contains user defined
variables. */

int     getop(char []);
void    push(double);
double pop(void);
double top(void);
int     clear(void);
int     swap(void);
int     elem(void);
int     dup(void);
void    sprnt(void);
void    result(void);
void    set_solution(void);
void    print_help(void);

int main()
{
        int type;
        int i, j;
```

```
        int op3;
        double topd;
        double op2;
        char s[MAXOP];
        char tmp[MAXOP];

        for (i = 0; i < VARMAX; i++) {
                var_array[i] = 0;
        }

        print_help();

        while ((type = getop(s)) != EOF) {

                op3 = elem();
                if (op3 == 0) { /* Only one input
completed. */
                        is_first_input = 1;
                } else if (op3 > 1) {
                        is_first_input = 0;
                }

                i = j = 0;

                switch (type) {
                case NUMBER:
                        push(atof(s));
                        break;
                case VARIABLE:
                        for (i = 2; s[i] != '\0'; i+
+){
                                tmp[j++] = s[i];
                                tmp[j] = '\0';
                        }
                        var_array[s[0] - 'A'] =
atof(tmp);
                        break;
                case '+':
                        push(pop() + pop());
                        break;
                case '*':
```

```
                        push(pop() * pop());
                        break;
                case '-':
                        op2 = pop();
                        push(pop() - op2);
                        break;
                case '/':
                        op2 = pop();
                        if (op2 != 0.0){
                                push(pop() / op2);
                        } else {
                                printf("Error: Divide
by zero.\n");
                        }
                        break;
                case '%':
                        op3 = (int) pop();
                        push((int) pop() % op3);
                        break;
                case 'c':
                        if (clear()) {
                                printf("Stack
Cleared.\n");
                        }
                        break;
                case 'p':
                        if ((topd = top()) != 0) {
                                printf("Top stack
element: %g", topd);
                                printf(" of %d
elements.\n", elem());
                        }
                        break;
                case 's':
                        if (swap()) {
                                printf("Swap
successful.\n");
                        }
                        break;
                case 'd':
                        if (dup()) {
```

```
                                        printf("Duplication
is successful.\n");
                                } else {
                                        printf("Error: Stack
empty.\n");
                                }
                                break;
                        case 'r':
                                sprnt();
                                break;
                        case 'o':
                                if (elem() < 2) {
                                        printf("Error: pow
requires at least two ");
                                        printf("items on the
stack.\n");

                                        break;
                                }
                                op2 = pop();
                                push(pow(op2, pop()));
                                break;
                        case 'i':
                                set_solution();
                                push(sin(pop()));
                                result();
                                break;
                        case 'y':
                                set_solution();
                                push(cos(pop()));
                                break;
                        case 't':
                                set_solution();
                                push(tan(pop()));
                                break;
                        case 'x':
                                set_solution();
                                push(exp(pop()));
                                break;
                        case 'q':
                                set_solution();
                                push(sqrt(pop()));
```

```
                        break;
                case 'f':
                        set_solution();
                        push(floor(pop()));
                        break;
                case 'l':
                        set_solution();
                        push(ceil(pop()));
                        break;
                case 'v':
                        for (i = 0; i < VARMAX; i++)
{
                        if (i < VARMAX-1) {
                                printf("%c: %10.10G
\n", 'A' + i, var_array[i]);
                        } else {
                                printf("%c: %10.10G
\n", '=', var_array[VARMAX]);
                        }
                        }
                        break;
                case 'h':
                        print_help();
                        break;
                case '\n':
                        result();
                        break;
                default:
                        if ((type >= 'A' && type <=
'Z') || type == '=') {
                                if (type != '=') {

push(var_array[type - 'A']);
                                } else {

push(var_array[VARMAX]);
                                }
                        } else {
                                printf("Error:
Unknown command \'%s\'\n", s);
                        }
```

```
                        break;

                }
        }
        return 0;
}

#define MAXVAL  100     /* Maximum depth of val
stack. */
int sp = 0;             /* Next free stack position.
*/
double val[MAXVAL];     /* Value stack. */

void push(double f)
{
        if (sp < MAXVAL) {
                val[sp++] = f;
        } else {
                printf("Error: Stack full, cannot
push %g\n", f);
        }
}

double pop(void)
{
        if (sp > 0) {
                return val[--sp];
        } else {
                printf("Error: Stack empty.\n");
                return 0.0;
        }
}

double top(void)
{
        if (sp > 0) {
                return val[sp-1];
        } else {
                printf("Error: Stack empty.\n");
                return 0.0;
        }
```

```
}

int clear(void)
{
        if (sp > 0) {
                while(val[--sp] != '\0');
                sp = 0;
                return 1;
        } else {
                printf("Error: Stack empty.\n");
                return 0;
        }
}

int swap(void)
{
        double sbuf;
        if (sp > 0) {
                sbuf = val[sp-2];
                val[sp-2] = val[sp-1];
                val[sp-1] = sbuf;
                return 1;
        } else {
                printf("Error: Stack empty.\n");
                return 0;
        }
}

int elem(void)
{
        return sp;
}

int dup (void)
{
        if (sp > 0) {
                sp++;
                val[sp] = val[sp-1];
                return 1;
        } else {
                return 0;
```

```
        }
}

void sprnt(void)
{
        int count = 0;
        while (count < sp) {
                printf("%d:%10.12g\n", count+1,
val[count]);
                count++;
        }
}

void result(void)
{
        if (sp == 1 && is_first_input != 1) {
                printf("Solution: %10.20g\n",
val[0]);
                var_array[VARMAX] = val[0];
                is_first_input = 0;
                clear();
        }
}

/*
 * Opens result() for execution.
 * Primarily used with the math functions because
they can be used with only
 * one stack item. For ex, if "1 i" is entered as the
first input, this
 * function would allow for a result to be shown and
the stack cleared.
 */
void set_solution(void)
{
        if (elem() >= 1) {
                is_first_input = 0;
        }
}

int getch(void);
```

```
void ungetch(int);

int getop(char s[])
{
        int i = 0, c;
        while ((s[0] = c = getch()) == ' ' || c ==
'\t');

        s[1] = '\0';

        if (isalpha(c) && c >= 'A' && c <= 'Z') {
                /* Collect the variable. */
                for ( ; s[i] != ' ' && s[i] != '\n';
s[++i] = getch());
                s[i] = '\0';

                if (i > 1) { /* A properly formed
variable definition. */
                        return VARIABLE;
                } else {
                        return c;
                }
        } else if (!isdigit(c) && c != '.' && c !=
'-') {
                return c; /* Not a number. */
        }

        if (c == '-') {
                if ((c = getch()) == ' ') {
                        /* If the next char is space,
then c is a operator. */
                        return c;
                } else if (isdigit(c)) {
                        s[++i] = c;
                }

        }

        if (isdigit(c)) { /* Collect integer part. */
                while (isdigit(s[++i] = c =
getch()));
```

```
        }

        if (c == '.') {   /* Collect fraction part.
*/
                while (isdigit(s[++i] = c =
getch()));
        }

        s[i] = '\0';
        if (c != EOF) {
                ungetch(c);
        }

        return NUMBER;
}

#define BUFSIZE 100

char buf[BUFSIZE];        /* Buffer for ungetch. */
int bufp = 0;             /* Next free position in buf.
*/

int getch(void)
{
        return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)
{
        if (bufp >= BUFSIZE) {
                printf("Ungetch: Too many characters.
\n");
        } else {
                buf[bufp++] = c;
        }
}

void print_help(void)
{
        printf("The Polish Calculator\n");
```

```
printf("-----------------------------------------------------
--\n");
        printf("-> Enter equations in the form: \"1 1
+ 2 5 + *\"\n");
        printf("-> Use \"A=1 B=2 C=3\" to store
variables.\n");
        printf("-> Use \"A B C * *\" to use stored
variables.\n");

printf("-----------------------------------------------------
--\n");
        printf(">>> Command Help:\n");
        printf(">>>     c:      Clear memory.\n");
        printf(">>>     p:      Print last character.
\n");
        printf(">>>     s:      Swap last two
characters.\n");
        printf(">>>     d:      Duplicate the last
input.\n");
        printf(">>>     r:      Print the entire
stack.\n");
        printf(">>>     v:      Print variable list.
\n");
        printf(">>>     o:      pow(x,y), x^y, x >
0.\n");
        printf(">>>     i:      sin(x), sine of x.
\n");
        printf(">>>     y:      cos(x), cosine of x.
\n");
        printf(">>>     t:      tan(x), tangent of x.
\n");
        printf(">>>     x:      exp(x), e^x,
exponential function.\n");
        printf(">>>     q:      sqrt(x), x >= 0,
square of x.\n");
        printf(">>>     f:      floor(x), largest
integer not greater than x.\n");
        printf(">>>     l:      ceil(x), smallest
integer not less than x.\n");
        printf(">>>     =:      Access the last
successful solution.\n");
```

```
        printf(">>>     h:      Print this help text.
\n");
}
```

solution2.better

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>

#define MAXOP 100
#define NUMBER 0
#define MAXVAL 100
#define BUFSIZE 100
#define IDENTIFIER 1
#define VARIABLE 2

int getop(char *);
void push(double);
double pop(void);
int getch(void);
void ungetch(int);
void viewstack(void);
void showTop(void);
void swap(void);
void duplicate(void);
void clearStack(void);
void mathfunc(char *);

char last;
double variableValue[26] = {0};

int bufp = 0;
char buf[BUFSIZE];
int sp = 0;
double val[MAXVAL];
```

```
void push(double f)
{
    if(sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n",
f);
}

double pop(void)
{
    if(sp > 0)
        return val[--sp];
    else
    {
        printf("error: stack empty\n");
        exit(1);
        //return 0.0;
    }
}

int getch(void)
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)
{
    if(bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}

int getop(char *s)
{
    int i, c, d, flag, len;
    while((s[0] = c = getch()) == ' ' || c == '\t');
    s[1] = '\0';
    if(!isalnum(c) && c != '.' && c != '-')
        return c;
```

```c
    if(c == '-')
    {
        d = getch();
        if(d == ' ')
            return c;
        else
            ungetch(d);
    }
    i = 0;
    if(isalnum(c) || c == '-')
        while(isalnum(s[++i] = c = getch()));
    if(c == '.')
        while(isalnum(s[++i] = c = getch()));
    s[i] = '\0';
    if(c != EOF)
        ungetch(c);
    flag = 1;
    len = strlen(s);
    if(len == 1 && isalpha(s[0]))
    {
        last = s[0];
        return VARIABLE;
    }
    for(i = 0; i < len; i++)
        if(!isalpha(s[i]))
        {
            flag = 0;
            break;
        }
    if(flag == 1)
        return IDENTIFIER;
    else
        return NUMBER;
}

void viewstack(void)
{
    int i;
    printf("\nstack:\n");
    for(i = sp - 1; i >= 0; i--)
        printf("%lf\n", val[i]);
```

```
}

void showTop(void)
{
    sp > 0 ? printf("\t%.8g\n", val[sp-1]) :
printf("stack is empty\n");
}

void swap(void)
{
    double temp;
    if(sp < 1)
        printf("error: stack has less than 2
elements, can't swap\n");
    else
    {
        temp = val[sp - 1];
        val[sp - 1] = val[sp - 2];
        val[sp - 2] = temp;
    }
}

void duplicate(void)
{
    if(sp > MAXVAL - 1)
        printf("error: stack is full, can't duplicate
\n");
    else
    {
        double temp = pop();
        push(temp);
        push(temp);
        ++sp;
    }
}

void clearStack(void)
{
    sp = 0;
}
```

```
void mathfunc(char *s)
{
    if(strcmp(s, "sin") == 0)
    {
        if(sp < 1)
            printf("error: stack is empty, can't use 
sin function\n");
        else
            push(sin(pop()));
    }
    else if(strcmp(s, "cos") == 0)
    {
        if(sp < 1)
            printf("error: stack is empty, can't use 
cos function\n");
        else
            push(cos(pop()));
    }
    else if(strcmp(s, "exp") == 0)
    {
        if(sp < 1)
            printf("error: stack is empty, can't use 
exp function\n");
        else
            push(exp(pop()));
    }
    else if(strcmp(s, "pow") == 0)
    {
        if(sp < 2)
            printf("error: stack has less than 2 
elements, can't use pow function\n");
        else
        {
            double op2;
            op2 = pop();
            push(pow(pop(), op2));
        }
    }
    else
        printf("%s is not a supported function\n", 
s);
```

```
}

int main()
{
    int type;
    double op2;
    char s[MAXOP];

    while((type = getop(s)) != EOF)
    {
        //viewstack(); // Use this function if you
wish to see the stack after every iteration
        switch(type)
        {
            case NUMBER:
                push(atof(s));
                break;
            case IDENTIFIER:
                mathfunc(s);
                break;
            case VARIABLE:
                push(variableValue[last - 97]);
                break;
            case '_': // This prints the most recent
variable
                push(0); // So that when \n is
encountered next, the stack isn't empty
                printf("%c\n", last);
                break;
            case '+':
                push(pop() + pop());
                break;
            case '*':
                push(pop() * pop());
                break;
            case '-':
                op2 = pop();
                push(pop() - op2);
                break;
            case '/':
                op2 = pop();
```

```
                if(op2 != 0.0)
                    push(pop() / op2);
                else
                    printf("error: zero divisor\n");
                break;
            case '%':
                op2 = pop();
                if(op2 != 0.0)
                    push(fmod(pop(), op2));
                else
                    printf("error: division by zero
\n");
                break;
            case '?': // show top item on stack
without popping
                showTop();
                break;
            case '~': // swap top two elements of
the stack
                swap();
                break;
            case '#': // duplicate the top element
                duplicate();
                break;
            case '!': // clearStack
                clearStack();
                break;
            case '\n':
                printf("\t%.8g\n", pop());
                break;
            default:
                printf("error: unknown command %s
\n", s);
                break;
        }
        //viewstack();
    }
    return 0;
}
```

7. *Write a routine* `ungets(s)` *that will push back an entire string onto the input. Should* `ungets` *know about* `buf` *and* `bufp` *, or should it just use* `ungetch` *?*

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

#define BUFSIZE 100

char buf[BUFSIZE]; /* buffer for ungetch */
int bufp = 0; /* next free position in buf */

int getch(void) /* get a (possibly pushed back)
character */
{
  return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c) /* push character back on input
*/
{
  if(bufp >= BUFSIZE)
    printf("ungetch: too many characters\n");
  else
    buf[bufp++] = c;
}

/*
   ungets() actually takes a little bit of thought.
Should the
   first character in "s" be sent to ungetch() first,
or should
   it be sent last?  I assumed that most code calling
getch()
   would be of this form:

     char array[...];
     int i;

     while (...) {
```

```
        array[i++] = getch();
      }

    In such cases, the same code might call ungets()
as:

      ungets(array);

    and expect to repeat the while loop to get the
same string
    back.  This requires that the last character be
sent first
    to ungetch() first, because getch() and ungetch()
work with
    a stack.

    To answer K&R2's additional question for this
problem,
    it's usually preferable for something like
ungets() to just
    build itself on top of ungetch().  This allows us
to change
    ungetch() and getch() in the future, perhaps to
use a linked
    list instead, without affecting ungets().
*/
void ungets(const char *s)
{
  size_t i = strlen(s);

  while (i > 0)
    ungetch(s[--i]);
}

int main(void)
{
  char *s = "hello, world.  this is a test.";
  int c;

  ungets(s);
  while ((c = getch()) != EOF)
```

```
    putchar(c);
  return 0;
}
```

8. *Suppose there will never be more than one character of pushback. Modify `getch` and `ungetch` accordingly.*

```
#include <stdio.h>

int buf = EOF; /* buffer for ungetch */

int getch(void) /* get a (possibly pushed back) character */
{
  int temp;

  if (buf != EOF) {
    temp = buf;
    buf = EOF;
  } else {
    temp = getchar();
  }
  return temp;
}

void ungetch(int c) /* push character back on input */
{
  if(buf != EOF)
    printf("ungetch: too many characters\n");
  else
    buf = c;
}

int main(void)
{
  int c;

  while ((c = getch()) != EOF) {
```

```
    if (c == '/') {
      putchar(c);
      if ((c = getch()) == '*') {
        ungetch('!');
      }
    }
    putchar(c);
  }
  return 0;
}
```

9. *Our `getch` and `ungetch` do not handle a pushed-back `EOF` correctly. Decide what their properties ought to be if an `EOF` is pushed back, then implement your design.*

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

int     getch    (void);
void    ungetch  (int);

/*
 * In the case that EOF is pushed back to ungetch(),
the program will simply
 * continue execution as it normally would. Ignoring
the EOF.
 */
int main()
{
        char c;

        /* Prematurely send EOF. */
        ungetch(EOF);

        while ((c = getch()) != EOF) {
                putchar(c);
        };

        return 0;
}
```

```
#define BUFSIZE 100
char buf[BUFSIZE];          /* Buffer for ungetch. */
int bufp = 0;               /* Next free position in buf.
*/

int getch(void)
{
        return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)
{
        if (bufp >= BUFSIZE) {
                printf("Ungetch: Too many characters.
\n");
        /*
         * The check for EOF must be made here. If
EOF is found, it is ignored.
         */
        } else if (c != EOF) {
                buf[bufp++] = c;
        }
}
```

10. *An alternate organization uses* `getline` *to read an entire input line; this makes* `getch` *and* `ungetch` *unnecessary. Revise the calculator to use this approach.*

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

#define MAXOP     100        /* Max size of operand
or operator. */
#define NUMBER    '0'        /* Signal that a number
was found. */
#define VARIABLE  '1'
#define VARMAX     27
```

```c
#define BUFSIZE 1000

int is_first_input = 0;     /* Prevents the solution
from being printed on first
                              input */
double var_array[VARMAX];   /* Contains user defined
variables. */

int    getop(char []);
void   push(double);
double pop(void);
double top(void);
int    clear(void);
int    swap(void);
int    elem(void);
int    dup(void);
void   sprnt(void);
void   result(void);
void   set_solution(void);
void   print_help(void);
int    mygetline(char [], int);

int main()
{
        int type;
        int i, j;
        int op3;
        double topd;
        double op2;
        char ic[MAXOP]; /* Input Char */
        char tmp[MAXOP];

        for (i = 0; i < VARMAX; i++) {
                var_array[i] = 0;
        }

        print_help();

        while ((type = getop(ic)) != EOF) {

                op3 = elem();
```

```
                if (op3 == 0) { /* Only one input
completed. */
                        is_first_input = 1;
                } else if (op3 > 1) {
                        is_first_input = 0;
                }

                i = j = 0;

                switch (type) {
                case NUMBER:
                        push(atof(ic));
                        break;
                case VARIABLE:
                        for (i = 2; ic[i] != '\0'; i+
+){
                                tmp[j++] = ic[i];
                                tmp[j] = '\0';
                        }
                        var_array[ic[0] - 'A'] =
atof(tmp);
                        break;
                case '+':
                        push(pop() + pop());
                        break;
                case '*':
                        push(pop() * pop());
                        break;
                case '-':
                        op2 = pop();
                        push(pop() - op2);
                        break;
                case '/':
                        op2 = pop();
                        if (op2 != 0.0){
                                push(pop() / op2);
                        } else {
                                printf("Error: Divide
by zero.\n");
                        }
                        break;
```

```
                case '%':
                        op3 = (int) pop();
                        push((int) pop() % op3);
                        break;
                case 'c':
                        if (clear()) {
                                printf("Stack
Cleared.\n");
                        }
                        break;
                case 'p':
                        if ((topd = top()) != 0) {
                                printf("Top stack
element: %g", topd);
                                printf(" of %d
elements.\n", elem());
                        }
                        break;
                case 's':
                        if (swap()) {
                                printf("Swap
successful.\n");
                        }
                        break;
                case 'd':
                        if (dup()) {
                                printf("Duplication
is successful.\n");
                        } else {
                                printf("Error: Stack
empty.\n");
                        }
                        break;
                case 'r':
                        sprnt();
                        break;
                case 'o':
                        if (elem() < 2) {
                                printf("Error: pow
requires at least two ");
                                printf("items on the
```

```
stack.\n");
                                break;
                        }
                        op2 = pop();
                        push(pow(op2, pop()));
                        break;
                case 'i':
                        set_solution();
                        push(sin(pop()));
                        result();
                        break;
                case 'y':
                        set_solution();
                        push(cos(pop()));
                        break;
                case 't':
                        set_solution();
                        push(tan(pop()));
                        break;
                case 'x':
                        set_solution();
                        push(exp(pop()));
                        break;
                case 'q':
                        set_solution();
                        push(sqrt(pop()));
                        break;
                case 'f':
                        set_solution();
                        push(floor(pop()));
                        break;
                case 'l':
                        set_solution();
                        push(ceil(pop()));
                        break;
                case 'v':
                        for (i = 0; i < VARMAX; i++)
{
                        if (i < VARMAX-1) {
                                printf("%c: %10.10G
\n", 'A' + i, var_array[i]);
```

```
                        } else {
                                printf("%c: %10.10G
\n", '=', var_array[VARMAX]);
                        }
                        }
                        break;
                case 'h':
                        print_help();
                        break;
                case '\n':
                        result();
                        break;
                default:
                        if ((type >= 'A' && type <=
'Z') || type == '=') {
                                if (type != '=') {

push(var_array[type - 'A']);
                                } else {

push(var_array[VARMAX]);
                                }
                        } else {
                                printf("Error:
Unknown command \'%s\'\n", ic);
                        }
                        break;

                }
        }
        return 0;
}

#define MAXVAL  100     /* Maximum depth of val
stack. */
int sp = 0;             /* Next free stack position.
*/
double val[MAXVAL];     /* Value stack. */

void push(double f)
{
```

```
        if (sp < MAXVAL) {
                val[sp++] = f;
        } else {
                printf("Error: Stack full, cannot
push %g\n", f);
        }
}

double pop(void)
{
        if (sp > 0) {
                return val[--sp];
        } else {
                printf("Error: Stack empty.\n");
                return 0.0;
        }
}

double top(void)
{
        if (sp > 0) {
                return val[sp-1];
        } else {
                printf("Error: Stack empty.\n");
                return 0.0;
        }
}

int clear(void)
{
        if (sp > 0) {
                while(val[--sp] != '\0');
                sp = 0;
                return 1;
        } else {
                printf("Error: Stack empty.\n");
                return 0;
        }
}

int swap(void)
```

```
{
        double sbuf;
        if (sp > 0) {
                sbuf = val[sp-2];
                val[sp-2] = val[sp-1];
                val[sp-1] = sbuf;
                return 1;
        } else {
                printf("Error: Stack empty.\n");
                return 0;
        }
}

int elem(void)
{
        return sp;
}

int dup (void)
{
        if (sp > 0) {
                sp++;
                val[sp] = val[sp-1];
                return 1;
        } else {
                return 0;
        }
}

void sprnt(void)
{
        int count = 0;
        while (count < sp) {
                printf("%d:%10.12g\n", count+1,
val[count]);
                count++;
        }
}

void result(void)
{
```

```
        if (sp == 1 && is_first_input != 1) {
                printf("Solution: %10.20g\n",
val[0]);
                var_array[VARMAX] = val[0];
                is_first_input = 0;
                clear();
        }
}

/*
 * Opens result() for execution.
 * Primarily used with the math functions because
they can be used with only
 * one stack item. For ex, if "1 i" is entered as the
first input, this
 * function would allow for a result to be shown and
the stack cleared.
 */
void set_solution(void)
{
        if (elem() >= 1) {
                is_first_input = 0;
        }
}

int mygetline(char s[], int maxline)
{
        int i = 0;
        char c;
        while ((c = getchar()) != EOF && c != '\n' &&
i < maxline) {
                s[i++] = c;
        }
        if (c == '\n') {
                s[i++] = c;
        }
        s[i++] = '\0';
        return i;
}

char getopbuf[BUFSIZE];     /* A string buffer that
```

```
holds the mygetline. */
int  getopbufp   = 0;       /* Current position is
the string buffer. */
int  getopbuflen = 0;

int getop(char s[])
{
        int i = 0, c;
        s[0] = '\0';

        if (getopbuflen == 0 || getopbufp ==
getopbuflen-2) {
                getopbufp = 0;
                /* Get a new line of input from the
user. */
                getopbuflen = mygetline(getopbuf,
BUFSIZE);
        }

        if (getopbuf[getopbufp] == '\n') {
                return '\n';
        }

        /* Jump over spaces at the begining of the
string. */
        while (isspace(c = getopbuf[getopbufp++]));
        s[i++] = c;

        if (isalpha(c) && c >= 'A' && c <= 'Z') {
                /* Collect the variable. */
                for (i = 1; getopbuf[getopbufp] != '
'
                     && getopbuf[getopbufp] != '\n';
                     s[i++] = getopbuf[getopbufp++]);
                s[i] = '\0';
                if (i > 1) { /* A properly formed
variable definition. */
                        return VARIABLE;
                } else {
                        return c;
                }
```

```
        } else if (!isdigit(c) && c != '.' && c !=
'-') {
                return c; /* Not a number. */
        }

        if (c == '-') {
                if ((c = getopbuf[++getopbufp]) == '
') {
                        /* If the next char is space,
then c is a operator. */
                        return c;
                } else if (isdigit(c)) {
                        s[++i] = c;
                }

        }

        if (isdigit(c)) { /* Collect integer part. */
                for ( ; isdigit(c =
getopbuf[getopbufp]); i++, getopbufp++) {
                        s[i] = c;
                }
        }

        if (c == '.') {   /* Collect fraction part.
*/
                while (isdigit(s[i++] = c =
getopbuf[getopbufp++]));
        }

        s[i] = '\0';
        return NUMBER;
}

void print_help(void)
{
        printf("The Polish Calculator\n");

printf("-----------------------------------------
--\n");
        printf("-> Enter equations in the form: \"1 1
```

```
+ 2 5 + *\"\n");
        printf("-> Use \"A=1 B=2 C=3\" to store
variables.\n");
        printf("-> Use \"A B C * *\" to use stored
variables.\n");

printf("-----------------------------------------------------
--\n");
        printf(">>> Command Help:\n");
        printf(">>>     c:      Clear memory.\n");
        printf(">>>     p:      Print last character.
\n");
        printf(">>>     s:      Swap last two
characters.\n");
        printf(">>>     d:      Duplicate the last
input.\n");
        printf(">>>     r:      Print the entire
stack.\n");
        printf(">>>     v:      Print variable list.
\n");
        printf(">>>     o:      pow(x,y), x^y, x >
0.\n");
        printf(">>>     i:      sin(x), sine of x.
\n");
        printf(">>>     y:      cos(x), cosine of x.
\n");
        printf(">>>     t:      tan(x), tangent of x.
\n");
        printf(">>>     x:      exp(x), e^x,
exponential function.\n");
        printf(">>>     q:      sqrt(x), x >= 0,
square of x.\n");
        printf(">>>     f:      floor(x), largest
integer not greater than x.\n");
        printf(">>>     l:      ceil(x), smallest
integer not less than x.\n");
        printf(">>>     =:      Access the last
successful solution.\n");
        printf(">>>     h:      Print this help text.
\n");
}
```

11. *Modify getop so that it doesn't need to use ungetch. Hint: use an internal static variable.*

```
#include <stdio.h>
#define NUMBER '0'

int getop(char *s)
{
    int c;
    static int buf = EOF;

    if (buf != EOF && buf != ' ' && buf != '\t'
        && !isdigit(buf) && buf != '.') {
        c = buf;
        buf = EOF;
        return c;
    }
    if (buf == EOF || buf == ' ' || buf == '\t')
        while ((*s = c = getch()) == ' ' || c ==
'\t')
            ;
    else
        *s = c = buf;
    buf = EOF;
    *(s + 1) = '\0';
    if (!isdigit(c) && c != '.')
        return c;       /* not a number */
    if (isdigit(c))     /* collect integer part */
        while (isdigit(*++s = c = getch()))
            ;
    if (c == '.')       /* collect fraction part */
        while (isdigit(*++s = c = getch()))
            ;
    *++s = '\0';
    buf = c;
    return NUMBER;
}
```

12. *Adapt the ideas of `printd` to write a recursive version of `itoa` ; that is, convert an integer into a string by calling a recursive routine.*

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define MAXSTRING 1000

void my_itoa(int n, char s[])
{
    static int i;
    static int sign = 0;


    if (sign == 0)  /* Get sign only if this was a
non-recursive call (called by a routine other than
itself).*/
    {
        sign = (n < 0) ? -1 : 1;
        i = 0; /* Index is initialized only on non-
recursive call. */
    }
    if (n / 10)
        my_itoa(n / 10, s);

    if (sign != 0)
    {
        if (sign < 0)
            s[i++] = '-';
        sign = 0; /* Reset the sign to get ready for
next non-recursive call. */
    }

    s[i++] = abs(n % 10) + '0';
    s[i] = '\0';
}

main()

{
    int array[22] =
    {   0,
        1,
```

```
        2,
        9,
        10,
        11,
        16,
        17,
        21,
        312,
        -0,
        -1,
        -2,
        -9,
        -10,
        -11,
        -16,
        -17,
        -21,
        -312,
        INT_MAX,
        INT_MIN,
        };

    int i;
    char s[MAXSTRING];

    for (i = 0; i < 22; ++i)
    {
        my_itoa(array[i], s);
        printf("%d    %s\n", array[i], s);
    }


    return 0;
}
```

13. *Write a recursive version of the function* `reverse(s)` *, which reverses the string* `s` *in place.*

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

void reverse(char [], int);
```

```c
int main()
{
        char tstr[] = "Hello, world!";
        printf("%s\n", tstr);
        reverse(tstr, 0);
        printf("%s\n", tstr);
        return 0;
}

/* I don't really like having the "left" argument,
but it is all I
 * could come up with without putting too much effort
into the problem.
 * and remaining a category 0 solution. */
void reverse(char s[], int left)
{
        int slen = strlen(s)-1; /* -1 is to
compensate for \0. */
        char buf = s[left];
        if (left < slen) {
                reverse(s, left+1);
        }
        if (buf != '\0') {
                s[slen-left] = buf;
        }
        if (left == 0) {
                /* Once execution reaches this point,
it is at the end of the
                 * first recursion and the
terminating char must be set to
                 * close the string.
                 */
                s[slen+1] = '\0';
        }
}
```

14. *Define a macro* `swap(t,x,y)` *that interchanges two arguments of type* `t`*. (Block structure will help.)*

```c
/* *Define a macro swap(t,x,y) that interchanges two
arguments of type t.
 *  (Block structure will help.)
 *
 * Feel free to modify and copy, if you really must,
but preferably not.
 * This is just an exercise in preprocessor
mechanics, not an example of
 * how it should really be used. The trickery is not
worth it to save three
 * lines of code.
 *
 * To exchange the values of two variables we need a
temporary variable and
 * this one needs a name. Any name we pick, the user
of the macro might also
 * use. Thus, we use the preprocessor argument
concatenation operator ## to
 * create the name from the actual variable names in
the call. This guarantees
 * that the result won't be either of the actual
arguments. In order to
 * make sure the result also does not fall into the
implementation's name
 * space, we prefix the name with something safe.
 *
 * Lars Wirzenius <liw@iki.fi>
 */

#include <stdio.h>

#define swap(t, x, y)     do {          t safe ## x ##
y;       safe ## x ## y = x;       x = y;       y =
safe ## x ## y;  } while (0)

int main(void) {
    int ix, iy;
    double dx, dy;
    char *px, *py;

    ix = 42;
```

```
    iy = 69;
    printf("integers before swap: %d and %d\n", ix,
iy);
    swap(int, ix, iy);
    printf("integers after swap: %d and %d\n", ix,
iy);

    dx = 123.0;
    dy = 321.0;
    printf("doubles before swap: %g and %g\n", dx,
dy);
    swap(double, dx, dy);
    printf("integers after swap: %g and %g\n", dx,
dy);

    px = "hello";
    py = "world";
    printf("pointers before swap: %s and %s\n", px,
py);
    swap(char *, px, py);
    printf("integers after swap: %s and %s\n", px,
py);

    return 0;
}
```

chapter 5.

1. *As written, `getint` treats a `+` or `-` not followed by a digit as a valid representation of zero. Fix it to push such a character back on the input.*

```
#include <ctype.h>
#include <stdio.h>
#define BUFFERLENGTH 100
int getch(void);
void ungetch(int);

int buf[BUFFERLENGTH];
int nfp = 0;
```

```
void viewbuffer(void)
{
    int i;
    printf("\nbuffer:\n");
    for(i = nfp - 1; i >= 0; i--)
        printf("%d\n", buf[i]);
}

int getch(void)
{
    return (nfp > 0) ? buf[--nfp] : getchar();
}

void ungetch(int c)
{
    if(nfp < BUFFERLENGTH)
        buf[nfp++] = c;
    else
        printf("error: ungetch buffer overflow\n");
}

int getint(int *pn)
{
    int c, sign;
    while(isspace(c = getch()));
    if(!isdigit(c) && c != EOF && c != '+' && c !=
'-')
    {
            ungetch(c);
            return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if(c == '+' || c == '-') // If character read is
+ or -
    {
        c = getch(); // Read next character
        if(!isdigit(c)) // If it is not a digit
        {
            ungetch(sign == -1 ? '-' : '+'); //
pushback on to the buffer the operator that was
```

```
previously read
            return 0;
        }
    }
    for(*pn = 0; isdigit(c); c = getch())
        *pn = *pn * 10 + (c - '0');
    *pn *= sign;
    if(c != EOF)
        ungetch(c);
    return c;
}

int main()
{
    int x, retval;
    int* px;
    px = &x;
    retval = getint(px);
    printf("retval = %d, x = %d\n", retval, x);
    viewbuffer();
    return 0;
}
```

2. *Write* `getfloat` *, the floating-point analog of* `getint` *. What type does* `getfloat` *return as its function value?*

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

/************ Input Sanitization Routine
******************/
#define SIZE 1000
int sanitizedInput[SIZE];
int len = 0;

void getSanitizedInput(void)
{
    int i, j, c;
    while((c = getchar()) != EOF) // Read all
```

```
characters from stdin and neglect anything that is
not a . + - or space.
    {
        if(isdigit(c) || c == '.' || c == '+' || c ==
'-' || c == ' ')
            sanitizedInput[len++] = c;
    }

    for(i = 0; i < len-1; i++) // Combinations such
as ++, +-, -+, --, .+, .-, .. etc are not allowed
    {
        if( (sanitizedInput[i] == '+' ||
sanitizedInput[i] == '-' || sanitizedInput[i] == '.')
&& (sanitizedInput[i+1] == ' ' || sanitizedInput[i+1]
== '+' || sanitizedInput[i+1] == '-' ||
sanitizedInput[i+1] == '.'))
        {
            if(!((sanitizedInput[i] == '+' &&
sanitizedInput[i+1] == '.') || (sanitizedInput[i] ==
'-' && sanitizedInput[i+1] == '.'))) // But +. and -.
are allowed
                sanitizedInput[i] = sanitizedInput[i
+1] = -1;
        }
    }

    j = 0;
    for(i = 0; i < len; i++)
        if(sanitizedInput[i] != -1)
            sanitizedInput[j++] = sanitizedInput[i];
    len = j;
}

void viewSanitizedInput(void)
{
    int i;
    for(i = 0; i < len; i++)
        printf("%c", sanitizedInput[i]);
    printf("\n");
}
```

```
/
******************************************************
*****/

/************************* main()
begins*****************/
int main()
{
    getSanitizedInput();
    //viewSanitizedInput();

    int n = 0;
    double ar[SIZE];
    for(n = 0; n < SIZE && getfloat(&ar[n]) != EOF; n
++)
        printf("ar[%d] = %lf\n", n, ar[n]);
    return 0;
}
/************************* main() ends
*******************/

/****************** xgetch()
function*********************/
int pos = 0;
int xgetch(void)
{
    return pos < len ? sanitizedInput[pos++] :
EOF; // Read from array storing the sanitized input,
if finished reading all elements then return EOF
}
/
******************************************************
*****/

/******************** getfloat() routine
*****************/
// The structure of the getfloat() function becomes
simplified because all spurious input has been
eliminated.
int getfloat(double *fp)
{
```

```
    int c, sign, flag = 0, k = 0;
    while(isspace(c = xgetch()));
    if(c == EOF)
        return c;
    sign = (c == '-') ? -1 : 1;
    if(c == '+' || c == '-')
        c = xgetch();
    for(*fp = 0; isdigit(c) || c == '.'; c =
xgetch())
    {
        if (c == '.')
            flag = 1;
        else
        {
            *fp = 10 * *fp + c - '0';
            if(flag == 1)
                k++;
        }
    }
    *fp = sign * *fp/pow(10, k);
}
/
*****************************************************
*****/
/*
On executing the above program with the following
input:

<file begins>
a    +58.479
b   *#$    -874.2154
c   \\\   .. 657.11258
d  ++ + - 254
e    (@)(^)  q---    -.247
f %^&#$ +.478
<file ends>

the output received is:

ar[0] = 58.479000
ar[1] = -874.215400
```

```
ar[2] = 657.112580
ar[3] = 254.000000
ar[4] = -0.247000
ar[5] = 0.478000
*/
```

3. *Write a pointer version of the function* `strcat` *that we showed in Chapter 2:* `strcat(s,t)` *copies the string* `t` *to the end of* `s` *.*

```
#include <stdio.h>

#define STR_BUFFER 10000

void strcat(char *, char *);

int main(int argc, char *argv[])
{
        char string1[STR_BUFFER] = "What A ";
        char string2[STR_BUFFER] = "Wonderful
World!";

        printf ("String 1: %s\n", string1);

        strcat(string1, string2);

        printf ("String 2: %s\n", string2);
        printf ("Cat Result: %s\n", string1);

        return 0;
}

/* Concatenate t to s. */
void strcat(char *s, char *t)
{
        /*
         * '*++s' is used to reference the pointer
before incrementing it so
         * that the check for falsehood ('\0') is
```

```
done with the next character
         * instead of '*s++' which would check, then
increment. Using '*s++'
         * would increment the pointer to the base
string past the null
         * termination character. When outputting the
string, this made it
         * appear that no concatenation occurred
because the base string is
         * cut off by the null termination character
('\0') that was never
         * copied over.
         */
        while(*s++); /* Get to the end of the string
*/
        s--;           /*get back to the end of the
string.*/
        while((*s++ = *t++));
}
```

4. *Write the function* `strend(s,t)` *, which returns 1 if the string* `t` *occurs at the end of the string* `s` *, and zero otherwise.*

```
#include <stdio.h>

int strend(char *s, char *t);

int main(void)
{
    char *s = "hello, hello";
    char *t = "llo";

    if (strend(s, t))
        printf("'%s' occurs at the end of '%s'\n", t,
s);
    else
        printf("no occurences at the end of '%s' from
'%s'\n", s, t);
```

```
    return 0;
}

int strend(char *s, char *t)
{
    int i = 0;
    int ind;

    while (*s) {
        for (ind = 0; *(s + ind) == *(t + ind)
                && *(s + ind) != '\0' && *(t +
ind) != '\0'; ind++)
            ;
        if(*(s + ind) == '\0' && *(t + ind) == '\0')
            return 1;
        s++;
    }

    return i;
}
```

5. *Write versions of the library functions* `strncpy` *,* `strncat` *, and* `strncmp` *, which operate on at most the first* `n` *characters of their argument strings. For example,* `strncpy(s,t,n)` *copies at most* `n`*characters of* `t` *to* `s` *. Full descriptions are in Appendix B.*

```
#include <stdio.h>

#define STR_BUF    10000
#define STR_MATCH  7 /* Used as the base number of
characters to match with. */

char *my_strncpy (char *, char *, int);
char *my_strncat (char *, char *, int);
int   my_strncmp (char *, char *, int);
int   my_strlen  (char *);

int main(int argc, char *argv[])
```

```
{
        int result;
        char str_s[STR_BUF] =  "All along the
watchtower.";
        char buf_1[STR_BUF];
        char buf_2[STR_BUF] = "Bob Dylan: ";
        char buf_3[STR_BUF] = "All along the
Watchposition.";

        printf
("-----------------------------------------------------
-------\n");
        printf ("  Base String: %s\n", str_s);
        printf
("-----------------------------------------------------
-------\n");

        my_strncpy (buf_1, str_s, STR_MATCH);
        printf ("buf_1 (my_strncpy, 7 chars): %s\n",
buf_1);

        my_strncat (buf_2, str_s, STR_MATCH);
        printf ("buf_2 (my_strncat, 5 chars): %s\n",
buf_2);

        result = my_strncmp(buf_3, str_s, STR_MATCH);

        printf ("buf_3 (my_strncmp, 6 chars): %s\n",
buf_3);

        if ( result == 0 ) {
                printf ("my_strncmp result: Both
strings match up to %d char(s).\n",
                        STR_MATCH );
        } else if ( result == -1 ) {
                printf ("my_strncmp result: Strings
do not match, buf_3 string ");
                printf ("has a lesser value.\n");
        } else if ( result == 1 ) {
                printf ("my_strncmp result: Strings
do not match, ");
```

```
                printf ("base string has a greater
value than buf_3.\n");
        }

        return 0;
}

/*
 * Copy at most n characters of string ct to s;
return s.
 */
char *my_strncpy (char *s, char *ct, int n)
{
        int count = 1;
        while ((*s++ = *ct++)) {
                if (count++ == n) {
                        break;
                }

        }
        return s;
}

/*
 * Concatenate at most n characters of string ct to
string s, terminate s with
 * '\0'; return s.
 */
char *my_strncat (char *s, char *ct, int n)
{
        int i = 0;
        int len = my_strlen(s);
        for (i = 0; n > 0; i++, n--) {
                *(s + len + i) = *ct++;
        }
        *(s + len + i) = '\0';
        return s;
}

/*
 * Compare at most n characters of string cs to
```

```
string ct; return < 0 if
 * cs < ct, 0 if cs == ct, or > 0 if cs > ct.
 */
int my_strncmp (char *cs, char *ct, int n)
{
        int i;

        for (i = 0; i < n; i++) {
                if (*(cs+i) < *(ct+i)) {
                        return -1;
                } else if (*(cs+i) > *(ct+i)) {
                        return 1;
                }
        }

        return 0;
}

int my_strlen (char *s)
{
        int count = 0;
        while (*s++ != '\0') {
                count++;
        }
        return count;
}
```

6. *Rewrite appropriate programs from earlier chapters and exercises with pointers instead of array indexing. Good possibilities include* `getline` *(Chapters 1 and 4),* `atoi` *,* `itoa` *, and their variants (Chapters 2, 3, and 4),* `reverse` *(Chapter 3), and* `strindex` *and* `getop` *(Chapter 4).*

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

#define STR_MAX   10000
#define BUFSIZE   100
```

```c
#define NUMBER    '0'

char *my_getline  (char *, int);
char *itoa     (int, char *);
int   atoi        (char *);
char *strrev      (char *);
int   strindex    (char *, char *);
int   getop       (char *);
int   getch       (void);
void  ungetch     (int);

int main(int argc, char *argv[])
{
        int num;
        char ss1[STR_MAX];
        char ss2[STR_MAX];
        char atoi_buf[STR_MAX];
        char itoa_buf[STR_MAX];
        int type;
        char s[BUFSIZE];

        printf (">>> Please enter a number: ");
        my_getline(atoi_buf, STR_MAX);
        num = atoi(atoi_buf);

        printf ("\natoi: str: %s, int: %d\n",
atoi_buf, num);
        printf ("itoa: int: %d, str: %s\n\n", num,
itoa(num, itoa_buf));

        printf (">>> Enter string 1: ");
        my_getline(ss1,STR_MAX);

        printf (">>> Enter string 2: ");
        my_getline(ss2,STR_MAX);

        printf ("\nThe reverse of string 1: %s\n",
strrev(ss1));
        printf ("The reverse of string 2: %s\n",
strrev(ss2));
```

```
        num = strindex(ss1, ss2);
        if (num == -1) {
                printf("\nThe substring (string 2)
was not found in the ");
                printf("base string (string 1).\n
\n");
        } else {
                printf("\nThe substring was found in
the base string, ");
                printf("starting at position %d.\n
\n", num);
        }

        printf (">>> Please enter a simple equation
(parsing example using getop()).\n");
        printf (">>> Example: 100+1039.238-acd\n");

        while ((type = getop(s)) != EOF) {
                switch (type) {
                case NUMBER:
                        printf ("Found a number: %s
\n", s);
                        break;
                case '+':
                        printf ("Found \'+\'\n");
                        break;
                case '\n':
                        printf ("Found Line Break.
\n");
                        break;
                default:
                        printf ("Found something
else: %s\n", s);
                        break;
                }
        }

        return 0;
}

char *my_getline(char *str, int str_max)
```

```
{
        char c;
        /*
         * A local variable to perform arithmetic on
so that 'str' points to
         * the correct location when it is passed
back to the caller.
         */
        char *s1 = str;
        while ((c = getchar()) != EOF && c != '\n' &&
str_max-- > 0 ) {
                *s1++ = c;
        }

        if (*s1 == '\n') {
                *s1++ = c;
        }

        *s1 = '\0';
        return str;
}

char *itoa(int num, char *str)
{
        char *ls = str;
        do
        {
                *ls++ = num % 10 + '0';
        } while ((num /= 10) > 0);
        strrev(str);
        return str;
}

int atoi(char *str)
{
        int n, sign;
        while (isspace(*str)) {
                str++;
        }
        sign = (*str == '-') ? -1 : 1;
        if (*str == '+' || *str == '-') {
```

```
                str++;
        }
        for (n = 0; isdigit(*str); str++) {
                n = 10 * n + (*str - '0');
        }
        return n * sign;
}

char *strrev(char *str)
{
        int i;
        char c;
        char *lp1 = str; /* The start of str. */
        char *lp2 = str; /* The end of str, for
incrementing.  */
        char *lp3 = str; /* The end of str, for
reference. */

        i =  strlen(str)-1;
        lp2 += i;
        lp3 += i;

        do
        {
                c = *lp1;
                *lp1++ = *lp2;
                *lp2-- = c;
        } while ((i -= 2) > 0);

        *++lp3 = '\0';
        return str;
}

int strindex(char *s, char *t)
{
        int i, j;
        char *sb = s;
        char *ss = s;
        char *tb = t;
        for (i = 0; *sb != '\0'; i++, sb++) {
                tb = t;  /* Reset the pointer to the
```

```
beginning of the string. */
                ss = sb; /* Reset the substring
pointer to the base string
                          * pointer. */
                for (j = 0; *tb != '\0' && *ss ==
*tb; ss++, tb++, j++) {
                        if (*(tb+1) == '\0' && j > 0)
{
                                return i;
                        }
                }
        }
        return -1;
}

int getop (char *str)
{
        int c;
        while ((*str++ = c = getch()) == ' ' || c ==
'\t');

        *str = '\0';
        if (!isdigit(c) && c != '.') {
                return c;
        }

        if (isdigit(c)) {
                while (isdigit(*str++ = c =
getch()));
        }

        if (c == '.') {         /* Collect fraction.
*/
                while (isdigit(*str++ = c =
getch()));
        }

        *--str = '\0';          /* Compensate for the
extra character. */
        if (c != EOF) {
                ungetch(c);     /* Return extra
```

```
charater to the stack. */
        }

        return NUMBER;
}

char buf[BUFSIZE];                /* The Stack */
int  bufp = 0;                    /* Top Position on
the stack */

int getch (void)
{
        return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch (int c)
{
        if (bufp >= BUFSIZE) {
                printf("ungetch: too many characters.
\n");
        } else {
                buf[bufp++] = c;
        }

}
```

7. *Rewrite* `readlines` *to store lines in an array supplied by* `main`*, rather than calling* `alloc` *to maintain storage. How much faster is the program?*

```
#include <stdio.h>
#include <string.h>
#define MAXLINES 5000
#define MAXLEN 1000
#define MAXSTORE 10000 /* max space allocated for all
lines.  Same as ALLOCSIZE on p.91. */


char *lineptr[MAXLINES];
char lines[MAXLINES][MAXLEN];
```

```
int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
int my_getline(char *, int);

void qsort(char *lineptr[], int left, int right);

int readlines2(char *lineptr[], int maxlines)
{
    int len, nlines;


    nlines = 0;
    while ((len = my_getline(lines[nlines], MAXLEN))
> 0)/*lines[nlines] is the address of the n-th
lines.*/
        if (nlines >= maxlines)
            return -1;
        else {
            lines[nlines][len - 1] = '\0';  /*
delete the newline */
            lineptr[nlines] = lines[nlines]; /*
track of the pointer to n-th lines.*/
            nlines++; /* increment n*/
        }
    return nlines;
}

main()
{
    int nlines;
    char linestore[MAXSTORE]; /* array for storing
all lines */
    /* myreadlines will pass an extra parameter
linestore for storing all the input lines */
    if ((nlines = readlines2(lineptr, MAXLINES)) >=
0)
    {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
```

```
    }
    else
    {
        printf("error: input too big to sort\n");
        return 1;
    }
}



void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}

/* K&R2 p97 */

void qsort(char *v[], int left, int right)
{
    int i, last;
    void swap(char *v[], int i, int j);
    if (left >= right)
        return;
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}

/* K&R2 p99 */

void swap(char *v[], int i, int j)
{
    char *temp;
    temp = v[i];
    v[i] = v[j];
```

```
    v[j] = temp;
}

/* K&R2 p29 */
int my_getline(char s[], int lim)
{
    int c, i;

    for (i = 0; i < lim - 1 && (c = getchar()) != EOF
&& c != '\n'; i++)
        s[i] = c;
    if (c == '\n') {
        s[i++] = c;
    }
    s[i] = '\0';
    return i;
}
```

8. *There is no error-checking in* `day_of_year` *or* `month_day`. *Remedy this defect.*

```
/*
 * A solution to exercise 5-8 in K&R2, page 112:
 *
 *   There is no error checking in day_of_year or
month_day. Remedy
 *   this defect.
 *
 * The error to check for is invalid argument values.
That is simple, what's
 * hard is deciding what to do in case of error. In
the real world, I would
 * use the assert macro from assert.h, but in this
solution I take the
 * approach of returning -1 instead. This is more
work for the caller, of
 * course.
 *
```

```
 * I have selected the year 1752 as the lowest
allowed year, because that
 * is when Great Britain switched to the Gregorian
calendar, and the leap
 * year validation is valid only for the Gregorian
calendar.
 *
 */

#include <stdio.h>

static char daytab[2][13] =  {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31},
};

/* day_of_year: set day of year from month & day */
int day_of_year(int year, int month, int day)
{
    int i, leap;

    if (year < 1752 || month < 1 || month > 12 || day
< 1)
        return -1;

    leap = (year%4 == 0 && year%100 != 0) || year%400
== 0;
    if (day > daytab[leap][month])
        return -1;

    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}

/* month_day: set month, day from day of year */
int month_day(int year, int yearday, int *pmonth, int
*pday)
{
```

```
    int i, leap;

    if (year < 1752 || yearday < 1)
        return -1;

    leap = (year%4 == 0 && year%100 != 0) || year%400
== 0;
    if ((leap && yearday > 366) || (!leap && yearday
> 365))
        return -1;

    for (i = 1; yearday > daytab[leap][i]; i++)
        yearday -= daytab[leap][i];
    *pmonth = i;
    *pday = yearday;

    return 0;
}


/* main: test day_of_year and month_day */
int main(void)
{
    int year, month, day, yearday;

    for (year = 1970; year <= 2000; ++year) {
        for (yearday = 1; yearday < 366; ++yearday) {
            if (month_day(year, yearday, &month,
&day) == -1) {
                printf("month_day failed: %d %d\n",
                    year, yearday);
            } else if (day_of_year(year, month,
day) != yearday) {
                printf("bad result: %d %d\n", year,
yearday);
                printf("month = %d, day = %d\n",
month, day);
            }
        }
    }
```

```
    return 0;
}
```

9. *Rewrite the routines* `day_of_year` *and* `month_day` *with pointers instead of indexing.*

```
#include <stdio.h>

static char daytab[2][13] =  {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30,
31},
};

/* original versions, for comparison purposes */

int day_of_year(int year, int month, int day)
{
    int i, leap;

    leap = (year%4 == 0 && year%100 != 0) || year%400
== 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}

void month_day(int year, int yearday, int *pmonth,
int *pday)
{
    int i, leap;

    leap = (year%4 == 0 && year%100 != 0) || year%400
== 0;
    for (i = 1; yearday > daytab[leap][i]; i++)
        yearday -= daytab[leap][i];
    *pmonth = i;
    *pday = yearday;
}
```

```c
/* pointer versions */

int day_of_year_pointer(int year, int month, int day)
{
    int i, leap;
    char *p;

    leap = (year%4 == 0 && year%100 != 0) || year%400
== 0;

    /* Set `p' to point at first month in the correct
row. */
    p = &daytab[leap][1];

    /* Move `p' along the row, to each successive
month. */
    for (i = 1; i < month; i++) {
        day += *p;
        ++p;
    }
    return day;
}

void month_day_pointer(int year, int yearday, int
*pmonth, int *pday)
{
    int i, leap;
    char *p;

    leap = (year%4 == 0 && year%100 != 0) || year%400
== 0;
    p = &daytab[leap][1];
    for (i = 1; yearday > *p; i++) {
        yearday -= *p;
        ++p;
    }
    *pmonth = i;
    *pday = yearday;
}
```

```c
int main(void)
{
    int year, month, day, yearday;

    year = 2000;
    month = 3;
    day = 1;
    printf("The date is: %d-%02d-%02d\n", year,
month, day);
    printf("day_of_year: %d\n", day_of_year(year,
month, day));
    printf("day_of_year_pointer: %d\n",
        day_of_year_pointer(year, month, day));


    yearday = 61;    /* 2000-03-01 */
    month_day(year, yearday, &month, &day);
    printf("Yearday is %d\n", yearday);
    printf("month_day: %d %d\n", month, day);
    month_day_pointer(year, yearday, &month, &day);
    printf("month_day_pointer: %d %d\n", month, day);

    return 0;
}
```

10. *Write the program* `expr` *, which evaluates a reverse Polish expression from the command line, where each operator or operand is a separate argument. For example,*
*expr 2 3 4 + **
*evaluates 2 X (3 + 4).*

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

#define NUMBER 0
```

```c
void push(double f);

double pop(void);


main(int argc, char *argv[])
{
    int type;
    int c;
    double op1, op2, latest;
    while (--argc > 0)
    {
        *++argv;
        if (!isdigit(c = **argv) && strlen(*argv) ==
1)
            type = c;
        else
            type = NUMBER;
        switch (type)
        {
            case NUMBER:
                push(atof(*argv));
                break;
            case '+':
                push(pop() + pop());
                break;
            case '*':
                push(pop() * pop());
                break;
            case '-':
                op2 = pop();
                push(pop() - op2);
                break;
            case '/':
                op2 = pop();
                if (op2 != 0.0)
                    push(pop() / op2);
                else
                    printf("error: zero divisor\n");
                break;
            case '%':
```

```
                op2 = pop();
                if (op2 != 0.0)
                     push(fmod(pop(), op2));
                else
                    printf("error: zero divisor\n");
                break;
            case '^':
                op2 = pop();
                op1 = pop();
                if (op1 == 0.0 && op2 <= 0)
                    printf("if x = 0.0, y must be
greater than 0\n");
                else
                    push(pow(op1, op2));
                break;
            case 'e':
                push(exp(pop()));
                break;
            case '~':
                push(sin(pop()));
                break;
            default:
                printf("error: unknown command: %c
\n", type);
                break;
        }
    }
    latest = pop();
    printf("\t%.8g\n", latest);
    return 0;
}

#define MAXVAL 100

int sp = 0;
double val[MAXVAL];
/* maximum depth of val stack */
/* next free stack position */
/* value stack */
/* push: push f onto value stack */
void push(double f)
```

```
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n",
f);
}
/* pop: pop and return top value from stack */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: stack empty\n");
        return 0.0;
    }
}
```

11. *Modify the programs* `entab` *and* `detab` *(written as exercises in Chapter 1) to accept a list of tab stops as arguments. Use the default tab settings if there are no arguments.*

```
/* include files */
#include <stdio.h>
#include <string.h>

/* macros */
#define NO_ARG          0
#define REQUIRED_ARG    1
#define OPTIONAL_ARG    2

/* types */

/* GETOPT_LONG_OPTION_T: The type of long option */
typedef struct GETOPT_LONG_OPTION_T {
    char *name;             /* the name of the long
option */
    int has_arg;            /* one of the above macros */
    int *flag;              /* determines if
getopt_long() returns a
```

```
                         * value for a long option;
if it is
                         * non-NULL, 0 is returned as
a function
                         * value and the value of val
is stored in
                         * the area pointed to by
flag.  Otherwise,
                         * val is returned. */
    int val;            /* determines the value to
return if flag is
                         * NULL. */
} GETOPT_LONG_OPTION_T;

typedef enum GETOPT_ORDERING_T {
    PERMUTE,
    RETURN_IN_ORDER,
    REQUIRE_ORDER
} GETOPT_ORDERING_T;

/* globally-defined variables */
char *optarg = NULL;
int optind = 0;
int opterr = 1;
int optopt = '?';

/* statically-defined variables */

static char *program_name;
 /* if nonzero, it means tab every x characters */
static unsigned long tab_every = 8;
 /* -i: only handle initial tabs/spaces */
static int flag_initial = 0;
 /* don't expand tabs into spaces */
static int flag_expand = 0;
static unsigned long *tab_stop_list = NULL;
static size_t num_tab_stops = 0;
static size_t num_tab_stops_allocked = 0;
static int show_help = 0;
static int show_version = 0;
static char *shortopts = "it:";
```

```
static GETOPT_LONG_OPTION_T longopts[] =
{
    {"initial", NO_ARG, NULL, 'i'},
    {"tabs", REQUIRED_ARG, NULL, 't'},

    {"help", NO_ARG, &show_help, 1},
    {"version", NO_ARG, &show_version, 1},
    {NULL, 0, 0, 0}
};

/* functions */

/* reverse_argv_elements:  reverses num elements
starting at argv */
static void reverse_argv_elements(char **argv, int
num)
{
    int i;
    char *tmp;

    for (i = 0; i < (num >> 1); i++) {
        tmp = argv[i];
        argv[i] = argv[num - i - 1];
        argv[num - i - 1] = tmp;
    }
}

/* permute: swap two blocks of argv-elements given
their lengths */
static void permute(char **argv, int len1, int len2)
{
    reverse_argv_elements(argv, len1);
    reverse_argv_elements(argv, len1 + len2);
    reverse_argv_elements(argv, len2);
}

/* is_option: is this argv-element an option or the
end of the option
list? */
static int is_option(char *argv_element, int only)
{
```

```
    return ((argv_element == NULL)
            || (argv_element[0] == '-')
            || (only && argv_element[0] == '+'));
}

/* getopt_internal:  the function that does all the
dirty work */
static int getopt_internal(int argc, char **argv,
char *shortopts,
                 GETOPT_LONG_OPTION_T * longopts, int
*longind, int
only)
{
    GETOPT_ORDERING_T ordering = PERMUTE;
    static size_t optwhere = 0;
    size_t permute_from = 0;
    int num_nonopts = 0;
    int optindex = 0;
    size_t match_chars = 0;
    char *possible_arg = NULL;
    int longopt_match = -1;
    int has_arg = -1;
    char *cp;
    int arg_next = 0;

    /* first, deal with silly parameters and easy
stuff */
    if (argc == 0 || argv == NULL || (shortopts ==
NULL && longopts == NULL))
        return (optopt = '?');
    if (optind >= argc || argv[optind] == NULL)
        return EOF;
    if (strcmp(argv[optind], "--") == 0) {
        optind++;
        return EOF;
    }
    /* if this is our first time through */
    if (optind == 0)
        optind = optwhere = 1;

    /* define ordering */
```

```
    if (shortopts != NULL && (*shortopts == '-' ||
*shortopts == '+')) {
        ordering = (*shortopts == '-') ?
RETURN_IN_ORDER : REQUIRE_ORDER;
        shortopts++;
    }
    else
        ordering = (getenv("POSIXLY_CORRECT") !=
NULL) ? REQUIRE_ORDER :
            PERMUTE;

    /* based on ordering, find our next option, if
we're at the beginning of
     * one
     */
    if (optwhere == 1) {
        switch (ordering) {
        case PERMUTE:
            permute_from = optind;
            num_nonopts = 0;
            while (!is_option(argv[optind], only)) {
                optind++;
                num_nonopts++;
            }
            if (argv[optind] == NULL) {
                /* no more options */
                optind = permute_from;
                return EOF;
            } else if (strcmp(argv[optind], "--") ==
0) {
                /* no more options, but have to get
`--' out of the way */
                permute(argv + permute_from,
num_nonopts, 1);
                optind = permute_from + 1;
                return EOF;
            }
            break;
        case RETURN_IN_ORDER:
            if (!is_option(argv[optind], only)) {
                optarg = argv[optind++];
```

```
                return (optopt = 1);
            }
            break;
        case REQUIRE_ORDER:
            if (!is_option(argv[optind], only))
                return EOF;
            break;
        }
    }
    /* we've got an option, so parse it */

    /* first, is it a long option? */
    if (longopts != NULL
        && (memcmp(argv[optind], "--", 2) == 0
            || (only && argv[optind][0] == '+'))
        && optwhere == 1) {
        /* handle long options */
        if (memcmp(argv[optind], "--", 2) == 0)
            optwhere = 2;
        longopt_match = -1;
        possible_arg = strchr(argv[optind] +
optwhere, '=');
        if (possible_arg == NULL) {
            /* no =, so next argv might be arg */
            match_chars = strlen(argv[optind]);
            possible_arg = argv[optind] +
match_chars;
            match_chars = match_chars - optwhere;
        }
        else
            match_chars = (possible_arg -
argv[optind]) - optwhere;
        for (optindex = 0; longopts[optindex].name !=
NULL; optindex++) {
            if (memcmp(argv[optind] + optwhere,
                       longopts[optindex].name,
                       match_chars) == 0) {
                /* do we have an exact match? */
                if (match_chars == (int)
(strlen(longopts[optindex].name))) {
                    longopt_match = optindex;
```

```
                    break;
                }
                /* do any characters match? */
                else {
                    if (longopt_match < 0)
                        longopt_match = optindex;
                    else {
                        /* we have ambiguous options
*/
                        if (opterr)
                            fprintf(stderr, "%s:
option `%s' is ambiguous "
                                    "(could be `--%s'
or `--%s')\n",
                                    argv[0],
                                    argv[optind],

longopts[longopt_match].name,

longopts[optindex].name);
                        return (optopt = '?');
                    }
                }
            }
        }
        if (longopt_match >= 0)
            has_arg =
longopts[longopt_match].has_arg;
    }
    /* if we didn't find a long option, is it a short
option? */
    if (longopt_match < 0 && shortopts != NULL) {
        cp = strchr(shortopts, argv[optind]
[optwhere]);
        if (cp == NULL) {
            /* couldn't find option in shortopts */
            if (opterr)
                fprintf(stderr,
                        "%s: invalid option -- `-
%c'\n",
                        argv[0],
```

```
                            argv[optind][optwhere]);
               optwhere++;
               if (argv[optind][optwhere] == '\0') {
                    optind++;
                    optwhere = 1;
               }
               return (optopt = '?');
          }
          has_arg = ((cp[1] == ':')
                         ? ((cp[2] == ':') ? OPTIONAL_ARG :
REQUIRED_ARG)
                         : NO_ARG);
          possible_arg = argv[optind] + optwhere + 1;
          optopt = *cp;
     }
     /* get argument and reset optwhere */
     arg_next = 0;
     switch (has_arg) {
     case OPTIONAL_ARG:
          if (*possible_arg == '=')
               possible_arg++;
          if (*possible_arg != '\0') {
               optarg = possible_arg;
               optwhere = 1;
          }
          else
               optarg = NULL;
          break;
     case REQUIRED_ARG:
          if (*possible_arg == '=')
               possible_arg++;
          if (*possible_arg != '\0') {
               optarg = possible_arg;
               optwhere = 1;
          }
          else if (optind + 1 >= argc) {
               if (opterr) {
                    fprintf(stderr, "%s: argument
required for option `",
                            argv[0]);
                    if (longopt_match >= 0)
```

```
                    fprintf(stderr, "--%s'\n",
longopts[longopt_match].name);
                else
                    fprintf(stderr, "-%c'\n", *cp);
            }
            optind++;
            return (optopt = ':');
        }
        else {
            optarg = argv[optind + 1];
            arg_next = 1;
            optwhere = 1;
        }
        break;
    case NO_ARG:
        if (longopt_match < 0) {
            optwhere++;
            if (argv[optind][optwhere] == '\0')
                optwhere = 1;
        }
        else
            optwhere = 1;
        optarg = NULL;
        break;
    }

    /* do we have to permute or otherwise modify
optind? */
    if (ordering == PERMUTE && optwhere == 1 &&
num_nonopts != 0) {
        permute(argv + permute_from, num_nonopts, 1 +
arg_next);
        optind = permute_from + 1 + arg_next;
    }
    else if (optwhere == 1)
        optind = optind + 1 + arg_next;

    /* finally return */
    if (longopt_match >= 0) {
        if (longind != NULL)
            *longind = longopt_match;
```

```
        if (longopts[longopt_match].flag != NULL) {
            *(longopts[longopt_match].flag) =
longopts[longopt_match].val;
            return 0;
        }
        else
            return longopts[longopt_match].val;
    }
    else
        return optopt;
}

int getopt_long(int argc, char **argv, char
*shortopts,
                 GETOPT_LONG_OPTION_T * longopts, int
*longind)
{
    return getopt_internal(argc, argv, shortopts,
longopts, longind, 0);
}

void help(void)
{
    puts( "OPTIONS" );
    puts( "" );
    puts( "-i, --initial   When shrinking, make
initial spaces/tabs on a line tabs" );
    puts( "                and expand every other tab
on the line into spaces." );
    puts( "-t=tablist,     Specify list of tab stops.
Default is every 8 characters." );
    puts( "--tabs=tablist, The parameter tablist is a
list of tab stops separated by" );
    puts( "-tablist        commas; if no commas are
present, the program will put a" );
    puts( "                tab stop every x places,
with x being the number in the" );
    puts( "                parameter." );
    puts( "" );
    puts( "--help          Print usage message and
exit successfully." );
```

```
    puts( "" );
    puts( "--version       Print version information
and exit successfully." );
}

void version(void)
{
    puts( "entab - shrink spaces into tabs" );
    puts( "Version 1.0" );
    puts( "Written by Gregory Pietsch" );
}

/* allocate memory, die on error */
void *xmalloc(size_t n)
{
    void *p = malloc(n);

    if (p == NULL) {
        fprintf(stderr, "%s: out of memory\n",
program_name);
        exit(EXIT_FAILURE);
    }
    return p;
}

/* reallocate memory, die on error */
void *xrealloc(void *p, size_t n)
{
    void *s;

    if (n == 0) {
        if (p != NULL)
            free(p);
        return NULL;
    }
    if (p == NULL)
        return xmalloc(n);
    s = realloc(p, n);
    if (s == NULL) {
        fprintf(stderr, "%s: out of memory\n",
program_name);
```

```
        exit(EXIT_FAILURE);
    }
    return s;
}

/* Determine the location of the first character in
the string s1
 * that is not a character in s2.  The terminating
null is not
 * considered part of the string.
 */
char *xstrcpbrk(char *s1, char *s2)
{
    char *sc1;
    char *sc2;

    for (sc1 = s1; *sc1 != '\0'; sc1++)
        for (sc2 = s2;; sc2++)
            if (*sc2 == '\0')
                return sc1;
            else if (*sc1 == *sc2)
                break;
    return NULL;                /* terminating nulls
match */
}

/* compare function for qsort() */
int ul_cmp(const void *a, const void *b)
{
    unsigned long *ula = (unsigned long *) a;
    unsigned long *ulb = (unsigned long *) b;

    return (*ula < *ulb) ? -1 : (*ula > *ulb);
}

/* handle a tab stop list -- assumes param isn't NULL
*/
void handle_tab_stops(char *s)
{
    char *p;
    unsigned long ul;
```

```
    size_t len = strlen(s);

    if (xstrcpbrk(s, "0123456789,") != NULL) {
        /* funny param */
        fprintf(stderr, "%s: invalid parameter\n",
program_name);
        exit(EXIT_FAILURE);
    }
    if (strchr(s, ',') == NULL) {
        tab_every = strtoul(s, NULL, 10);
        if (tab_every == 0)
            tab_every = 8;
    }
    else {
        tab_stop_list = xrealloc(tab_stop_list,
              (num_tab_stops_allocked += len) *
(sizeof(unsigned long)));
        for (p = s; (p = strtok(p, ",")) != NULL; p =
NULL) {
            ul = strtoul(p, NULL, 10);
            tab_stop_list[num_tab_stops++] = ul;
        }
        qsort(tab_stop_list, num_tab_stops,
sizeof(unsigned long),
              ul_cmp);
    }
}

void parse_args(int argc, char **argv)
{
    int opt;

    do {
        switch ((opt = getopt_long(argc, argv,
shortopts, longopts, NULL))) {
        case 'i':                    /* initial */
            flag_initial = 1;
            break;
        case 't':                    /* tab stops */
            handle_tab_stops(optarg);
            break;
```

```
        case '?':                    /* invalid option */
            fprintf(stderr, "For help, type:\n\t%s --
help\n", program_name);
            exit(EXIT_FAILURE);
        case 1:
        case 0:
            if (show_help || show_version) {
                if (show_help)
                    help();
                if (show_version)
                    version();
                exit(EXIT_SUCCESS);
            }
            break;
        default:
            break;
        }
    } while (opt != EOF);
}

/* output exactly n spaces */
void output_spaces(size_t n)
{
    int x = n;                       /* assume n is small
*/

    printf("%*s", x, "");
}

/* get next highest tab stop */
unsigned long get_next_tab(unsigned long x)
{
    size_t i;

    if (tab_stop_list == NULL) {
        /* use tab_every */
        x += (tab_every - (x % tab_every));
        return x;
    }
    else {
        for (i = 0; i < num_tab_stops &&
```

```
tab_stop_list[i] <= x; i++);
        return (i >= num_tab_stops) ? 0 :
tab_stop_list[i];
    }
}

/* the function that does the dirty work */
void tab(FILE * f)
{
    unsigned long linelength = 0;
    int c;
    int in_initials = 1;
    size_t num_spaces = 0;
    unsigned long next_tab;

    while ((c = getc(f)) != EOF) {
        if (c != ' ' && c != '\t' && num_spaces > 0)
{
            /* output spaces and possible tabs */
            if (flag_expand
                || (flag_initial && !in_initials)
                || num_spaces == 1) {
                /* output spaces anyway */
                output_spaces(num_spaces);
                linelength += num_spaces;
                num_spaces = 0;
            }
            else
                while (num_spaces != 0) {
                    next_tab =
get_next_tab(linelength);
                    if (next_tab > 0 && next_tab <=
linelength + num_spaces) {
                        /* output a tab */
                        putchar('\t');
                        num_spaces -= (next_tab -
linelength);
                        linelength = next_tab;
                    }
                    else {
                        /* output spaces */
```

```
                        output_spaces(num_spaces);
                        linelength += num_spaces;
                        num_spaces = 0;
                    }
                }
        }
        switch (c) {
        case ' ':               /* space */
            num_spaces++;
            break;
        case '\b':              /* backspace */
            /* preserve backspaces in output;
decrement length for tabbing
             * purposes
             */
            putchar(c);
            if (linelength > 0)
                linelength--;
            break;
        case '\n':              /* newline */
            putchar(c);
            in_initials = 1;
            linelength = 0;
            break;
        case '\t':              /* tab */
            next_tab = get_next_tab(linelength +
num_spaces);
            if (next_tab == 0) {
                while ((next_tab =
get_next_tab(linelength)) != 0) {
                    /* output tabs */
                    putchar('\t');
                    num_spaces -= (next_tab -
linelength);
                    linelength = next_tab;
                }
                /* output spaces */
                output_spaces(num_spaces);
                num_spaces = 0;
                putchar('\t');
                linelength += num_spaces + 1;
```

```
            }
            else
                num_spaces = next_tab - linelength;
            break;
        default:
            putchar(c);
            in_initials = 0;
            linelength++;
            break;
        }
    }
}

int main(int argc, char **argv)
{
    int i;
    FILE *fp;
    char *allocked_argvs = xmalloc(argc + 1);
    char **new_argv = xmalloc((argc + 1) *
sizeof(char *));
    char *p;

    program_name = argv[0];
    memset(allocked_argvs, 0, argc + 1);
    for (i = 0; i < argc; i++) {
        p = argv[i];
        if (isdigit(p[1])) {
            new_argv[i] = xmalloc(strlen(p) + 2);
            sprintf(new_argv[i], "-t%s", p + 1);
            allocked_argvs[i] = 1;
        }
        else
            new_argv[i] = p;
    }
    new_argv[argc] = NULL;
    parse_args(argc, new_argv);
    if (optind == argc)
        tab(stdin);
    else {
        for (i = optind; i < argc; i++) {
            if (strcmp(argv[i], "-") == 0)
```

```
                fp = stdin;
            else {
                fp = fopen(argv[i], "r");
                if (fp == NULL) {
                    fprintf(stderr, "%s: can't
open %s\n",
                            argv[0], argv[i]);
                    abort();
                }
            }
            tab(fp);
            if (fp != stdin)
                fclose(fp);
        }
    }
    /* free everything we can */
    for (i = 0; i < argc; i++)
        if (allocked_argvs[i])
            free(new_argv[i]);
    free(allocked_argvs);
    if (tab_stop_list != NULL)
        free(tab_stop_list);
    return EXIT_SUCCESS;
}

/* END OF FILE entab.c */
```

12. *Extend* `entab` *and* `detab` *to accept the shorthand* `entab -m +n` *to mean tab stops every* `n` *columns, starting at column* `m`*. Choose convenient (for the user) default behavior.*

```
/*
 * detab.c (Exercise 5-12)
 * About    :   Replaces tabs in the input with the
proper number
 *              of blanks to space to the next tab
stop.
 * Usage    :   detab -m +n
 *              (tab stops every n columns, starting
at column m)
```

```c
 */

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

#define TABSIZE 4    /* default tab size */

int main(int argc, char *argv[])
{
    int c, pos, i;
    int tabsize[2] = {TABSIZE, TABSIZE};

    while (--argc > 0 && (**++argv == '-' || **argv
== '+'))
        switch (**argv)
        {
            case '-':
                if ((pos = atoi(*argv + 1)) > 0)
                    tabsize[0] = pos;
                break;
            case '+':
                if ((pos = atoi(*argv + 1)) > 0)
                    tabsize[1] = pos;
                break;
        }

    pos = 0;
    i = 0;
    while ((c = getchar()) != EOF)
    {
        if (c == '\t')
        {
            while (pos < tabsize[i])
            {
                putchar(' ');
                ++pos;
            }
            pos = 0;
            if (i < 1)
                i++;
        }
```

```
        else
        {
            putchar(c);

            if (c == '\n')
            {
                pos = 0;
                i = 0;
            }
            else /* normal character */
            {
                ++pos;

                if (pos == tabsize[i])
                {
                    pos = 0;
                    if (i < 1)
                        i++;
                }
            }
        }
    }

    return 0;
}
```

13.*Write the program* `tail`, *which prints the last* `n` *lines of its input. By default,* `n` *is 10, say, but it can be changed by an optional argument, so that prints the last* `n` *lines. The program should behave rationally no matter how unreasonable the input or the value of* `n`. *Write the program so it makes the best use of available storage; lines should be stored as in the sorting program of Section 5.6, not in a two-dimensional array of fixed size.*

*example :tail -n // n is line number*

```
/
*****************************************************
**************************/
/* the first thing that came to mind - take program
```

```
from Chapter 5.6 ,          */
/*   and return last N rows:
*/
/*         Creating "char allocbuf[]", on each string
call "p = alloc (len)",   */
/*         copy string to p and wrint lineptr[i] = p
(in array of pointers).    */
/*
*/
/* But . In exercise said about the "maximum memory
savin"! Lack the proposed  */
/*   method is that "allocbuf" spent irrationally.
Suppose it was filling, but */
/*   stdio has any strings. Why not "erase" all
string except                   */
/*   the last n (interest strings) and continue to
work? Simply shift            */
/*   the last n strings (by character) in the "start
allocbuf" (remember the    */
/*   difference = `diff`), and in array of pointer
shift interest pointers      */
/*   (last n) to the `diff` values.
*/
/
*****************************************************
**************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define DEFAULT_STRING_NUM 10
#define MAXLEN 1000
#define MAXLINES 5000

char *lineptr[MAXLINES];

int getline(char *, int);
char *smartalloc(int n, char *begin);
int readlines(char *linestack[], int tail_num);
void writelines(char *lineptr[], int tail_num, int
nlines);
```

```
void writespecific(char *lineptr[]);

int main(int argc, char *argv[])
{
    int num, nlines;

    if (argc == 3 && strcmp(*(argv+1),"-n") == 0) {
        num = atoi(*(argv+2));
        if (num < 1) {
            printf("usage: 'tail -n COUNT_LINES'
\n");
            return -1;
        }
        else if (num > MAXLINES) {
            printf("n must be in {1..%d}
\n",MAXLINES);
            return -1;
        }
    }
    else if (argc == 1) {
        num = DEFAULT_STRING_NUM;
    }
    else {
        printf("usage: 'tail -n COUNT_LINES' \n");
        return -1;
    }
    if ((nlines=readlines(lineptr, num)) >= 0) {
        num = (num > nlines) ? nlines : num ; /* if
in stdio lines less, than user want (num) */
        writelines(lineptr, num, nlines);
        return 0;
    } else {
        printf("error: input too big to tail\n");
        return -1;
    }
}

/* readlines: */
int readlines(char *lineptr[], int tail_num)
{
    int len;                        /* length string
```

```
getted from getline */
    int nlines = 0;             /* total count strings
( <= tail_num )*/
    char *p;                    /* pointer to current
free position (returned by smartalloc) */
    char *leftpos = NULL;       /* pointer to
beginning interest block (use in smartalloc) */
    char *rightpos = NULL;      /* pointer to end
interest block (calculate as p+len)  */
    char line[MAXLEN];          /* current string */

    while ((len = getline(line, MAXLEN)) > 0) {
        if ((p = smartalloc(len+1, leftpos)) == NULL)
{ /* if buffer overfull */
            return -1;
        }
        else {
            /* check is `memory moved`? if next
pointer not "next" */
            if (rightpos+1 > p) {
                /* moving interest pointers value */
                for (int i = 0; i < tail_num; i++)
                    lineptr[nlines-i-1] -= rightpos
- p + 1;
                    /* (rightpos - p + 1) - moving
'diff' (in pointers) */
            }
            /* copying string */
            line[len] = '\0'; /* delete \n in line
*/
            strcpy(p, line);
            lineptr[nlines++] = p;
            if (nlines <= tail_num)
                leftpos = lineptr[0]; /* first
element */
            else
                leftpos = lineptr[nlines-
tail_num]; /* first interest element */
            rightpos = p + len;
        }
    }
```

```
    return nlines;
}

void writelines(char *lineptr[], int tail_num, int
nlines)
{
    for (int i = 0; i < tail_num; i++)
        printf("%s\n",lineptr[nlines-tail_num+i]);
}


#define ALLOCSIZE 100

static char allocbuf[ALLOCSIZE];
static char *allocp = &allocbuf[0];

static void movemem(char *start);

/**
 * n - lenght of asking memory
 * *begin - pointer to beginning interest blocks
 */
char *smartalloc(int n, char *begin)
{
    /* if first calling, begin set to buffer-begin */
    begin = (begin == 0) ? allocbuf : begin; // begin
== 0 eq begin == NULL
    if (allocbuf + ALLOCSIZE - allocp >= n) {
        allocp += n;
        return allocp - n;
    }
    else {  /* buffer full */
        movemem(begin);
        if (allocbuf + ALLOCSIZE - allocp >= n) {
            allocp += n;
            return allocp - n;
        }
        else
            return 0; /* movemem does not solve
problem */
    }
```

```
}

/**
 * move important memory blocks (it start from
*start) to buffer-begin
 */
static void movemem(char *begin)
{
    if (begin > allocbuf && begin < allocp) {
        char *p = &allocbuf[0];
        while (begin < allocp)
            *p++ = *begin++;
        allocp = p;
    }
}


/**
 * In difference with getline-KnR:
 * this function cut '\n'
 */
int getline(char s[], int lim)
{
    int i, c;

    for (i = 0; i<lim-1 && (c=getchar())!= EOF && c!
='\n'; i++)
        s[i] = c;
    s[i] = '\0';
    return i;
}
```

14. *Modify the sort program to handle a `-r` flag, which indicates sorting in reverse (decreasing) order. Be sure that `-r` works with `-n`.*

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

#define TRUE 1
#define FALSE 0
```

```
#define MAXLINES 5000       /* maximum number of
lines */
char *lineptr[MAXLINES];

#define MAXLEN 1000         /* maximum length of a
line */

int reverse = FALSE;

/* K&R2 p29 */
int getline(char s[], int lim)
{
  int c, i;

  for (i = 0; i < lim - 1 && (c = getchar()) != EOF
&& c != '\n'; i++)
    s[i] = c;
  if (c == '\n') {
    s[i++] = c;
  }
  s[i] = '\0';
  return i;
}

/* K&R2 p109 */
int readlines(char *lineptr[], int maxlines)
{
  int len, nlines;
  char *p, line[MAXLEN];

  nlines = 0;
  while ((len = getline(line, MAXLEN)) > 0)
    if (nlines >= maxlines || (p = malloc(len)) ==
NULL)
      return -1;
    else {
      line[len - 1] = '\0';  /* delete the newline */
      strcpy(p, line);
      lineptr[nlines++] = p;
    }
```

```c
  return nlines;
}

/* K&R2 p109 */
void writelines(char *lineptr[], int nlines)
{
  int i;

  for (i = 0; i < nlines; i++)
    printf("%s\n", lineptr[i]);
}

int pstrcmp(const void *p1, const void *p2)
{
  char * const *s1 = reverse ? p2 : p1;
  char * const *s2 = reverse ? p1 : p2;

  return strcmp(*s1, *s2);
}

int numcmp(const void *p1, const void *p2)
{
  char * const *s1 = reverse ? p2 : p1;
  char * const *s2 = reverse ? p1 : p2;
  double v1, v2;

  v1 = atof(*s1);
  v2 = atof(*s2);
  if (v1 < v2)
    return -1;
  else if (v1 > v2)
    return 1;
  else
    return 0;
}

int main(int argc, char *argv[])
{
  int nlines;
  int numeric = FALSE;
  int i;
```

```
  for (i = 1; i < argc; i++) {
    if (*argv[i] == '-') {
      switch (*(argv[i] + 1)) {
        case 'n':  numeric = TRUE;  break;
        case 'r':  reverse = TRUE;  break;
        default:
          fprintf(stderr, "invalid switch '%s'\n",
argv[i]);
          return EXIT_FAILURE;
      }
    }
  }

  if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
    qsort(lineptr, nlines, sizeof(*lineptr),
numeric ? numcmp : pstrcmp);
    writelines(lineptr, nlines);
    return EXIT_SUCCESS;
  } else {
    fputs("input too big to sort\n", stderr);
    return EXIT_FAILURE;
  }
}
```

15. *Add the option* `-f` *to fold upper and lower case together, so that case distinctions are not made during sorting; for example,* `a` *and* `A` *compare equal.*

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

#define MAXLINES 5000     /* max #lines to be sorted
*/
char *lineptr[MAXLINES];  /* pointers to text lines
*/

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);
```

```c
void my_qsort(void *lineptr[], int left, int right,
            int (*comp)(void *, void *), int order);

int numcmp(char *, char *);
int strcmp_f(char *, char *);

/* sort input lines */
int main(int argc, char *argv[])
{
    int nlines;         /* number of input lines read
*/
    int numeric = 0;    /* 1 if numeric sort */
    int reverse = 0;    /* 1 if sorting in reverse
order */
    int foldcase = 0;   /* 1 if sorting case
insensitive */

    while (--argc > 0)
    {
        if (strcmp(*++argv, "-n") == 0)
            numeric = 1;
        else if (strcmp(*argv, "-r") == 0)
            reverse = 1;
        else if (strcmp(*argv, "-f") == 0)
            foldcase = 1;
    }

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0)
    {
        my_qsort((void **) lineptr, 0, nlines-1,
            (int (*)(void *, void *))(numeric ?
numcmp : (foldcase ? strcmp_f : strcmp)),
            reverse ? -1 : 1);
        writelines(lineptr, nlines);
        return 0;
    }
    else
    {
        printf("input too big to sort\n");
        return 1;
```

```
    }

    return 0;
}

#define MAXLEN 1000  /* max length of any input line
*/

int getline(char *, int);
char *alloc(int);

/* readlines:  read input lines */
int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
    char *p, line[MAXLEN];

    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || (p = alloc(len)) ==
NULL)
            return -1;
        else
        {
            line[len-1] = '\0';  /* delete newline */
            strcpy(p, line);
            lineptr[nlines++] = p;
        }

    return nlines;
}

/* writelines:  write output lines */
void writelines(char *lineptr[], int nlines)
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

```
/* getline:  read a line into s, return length  */
int getline(char s[], int lim)
{
    int c, i;

    for (i = 0; i < lim-1 && (c = getchar()) != EOF
&& c != '\n'; ++i)
        s[i] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';

    return i;
}

#define ALLOCSIZE 10000 /* size of available space */

static char allocbuf[ALLOCSIZE]; /* storage for alloc
*/
static char *allocp = allocbuf;  /* next free
position */

char *alloc(int n)    /* return pointer to n
characters */
{
    if (allocbuf + ALLOCSIZE - allocp >= n)  /* it
fits */
    {
        allocp += n;
        return allocp - n;  /* old p */
    }
    else    /* not enough room */
        return 0;
}

#include <stdlib.h>

/* numcmp:  compare s1 and s2 numerically */
int numcmp(char *s1, char *s2)
{
    double v1, v2;
```

```
    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}

/* strcmp_f */
int strcmp_f(char *s, char *t)
{
    for ( ; toupper(*s) == toupper(*t); s++, t++)
        if (*s == '\0')
            return 0;

    return toupper(*s) - toupper(*t);
}

/* my_qsort:  sort v[left]...v[right] */
void my_qsort(void *v[], int left, int right,
            int (*comp)(void *, void *), int order)
{
    int i, last;

    void swap(void *v[], int, int);

    if (left >= right)    /* do nothing if array
contains */
        return;            /* fewer than two elements
*/
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right;  i++)
        if (order * (*comp)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    my_qsort(v, left, last-1, comp, order);
    my_qsort(v, last+1, right, comp, order);
```

```
}

void swap(void *v[], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

16. *Add the* `-d` *("directory order") option, which makes comparisons only on letters, numbers* and blanks. Make sure it works in conjunction with `-f` .

```
/*
 * Solution to K&R exercise 5-16
 * This solution uses a slightly modified version of
the readline function
 * that allows it to read very long lines.  However,
it can be used with
 * with the original readline.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

int getline(char*, int);
char *alloc(int);
int readlines(char**, int);
void writelines(char**, int);
void quicksort(void**, int, int, int (*)(void*,
void*));
int numcmp(char*, char*);
int mystrcmp(char*, char*);

#define MAXLINES 10000
#define MAXLEN 1000
```

```
char *lineptr[MAXLINES];  /* pointers to text lines
*/
int decreasing = 0;       /* 0 if increasing, 1 if
decreasing   -r flag */
int numeric = 0;          /* 1 if numeric sort   -n
flag */
int fold = 0;             /* 1 if not case-sensitive
-f flag */
int directory = 0;        /* 1 if directory sort   -d
flag */


int main(int argc, char* argv[])
{
    int nlines, i;

    while(--argc > 0) {
        ++argv;
        if((*argv)[0] == '-')
            for(i = 1; (*argv)[i]; ++i)
                switch((*argv)[i]) {
                    case 'n':
                        numeric = 1;
                        break;
                    case 'f':
                        fold = 1;
                        break;
                    case 'r':
                        decreasing = 1;
                        break;
                    case 'd':
                        directory = 1;
                        break;
                    default:
                        printf("usage: sort -dfnr
\n");
                        return 1;
                }
        else {
            printf("usage: sort -dfnr\n");
            return 1;
```

```c
        }
    }
    if((nlines = readlines(lineptr, MAXLINES)) >= 0)
{
        if(numeric)
            quicksort((void**) lineptr, 0, nlines -
1,
                      (int (*)(void*, void*))numcmp);
        else
            quicksort((void**) lineptr, 0, nlines -
1,
                      (int (*)(void*,
void*))mystrcmp);
        writelines(lineptr, nlines);
        return 0;
    }
    else {
        printf("input too big to sort\n");
        return 1;
    }
}

/* quicksort: sort v[left]...v[right] into increasing
or decreasing order */
void quicksort(void *v[], int left, int right, int
(*comp)(void*, void*))
{
    int i, last;
    void swap(void *v[], int, int);

    if(left >= right)        /* do nothing if array
contains */
        return;              /* fewer than two
elements      */
    swap(v, left, (left + right) / 2);  /* move
element to sort left */
    last = left;
    for(i = left + 1; i <= right; ++i) { /* move all
elements < or > sort */
        if(!decreasing) {                /* element
to the left according */
```

```
            if((*comp)(v[i], v[left]) < 0)  /* to
order */
                swap(v, ++last, i);
        }
        else
            if((*comp)(v[i], v[left]) > 0)
                swap(v, ++last, i);
    }
    swap(v, left, last);     /* move sort element to
its final position */
    quicksort(v, left, last - 1, comp);  /* sort left
subarray */
    quicksort(v, last + 1, right, comp); /* sort
right subarray */
}

/*
 * mystrcmp:  Compares s1 and s2 lexicographically.
Ignores characters that
 * are not letters, numbers, or whitespace if
directory flag is set.  If the
 * fold flag is set, it isn't case sensitive.
 */
int mystrcmp(char *s1, char *s2)
{
    if(directory) {
        while(!isdigit(*s1) && !isalpha(*s1) && !
isspace(*s1) && *s1)
            ++s1;       /* ignore bad characters */
        while(!isdigit(*s2) && !isalpha(*s2) && !
isspace(*s2) && *s2)
            ++s2;       /* ignore bad characters */
    }
    while(fold ? (tolower(*s1) == tolower(*s2)) :
(*s1 == *s2)) {
        if(*s1 == '\0')
            return 0;
        ++s1;
        ++s2;
        if(directory) {
            while(!isdigit(*s1) && !isalpha(*s1)
```

```
&& !isspace(*s1) && *s1)
                ++s1;       /* ignore bad characters
*/
            while(!isdigit(*s2) && !isalpha(*s2)
&& !isspace(*s2) && *s2)
                ++s2;       /* ignore bad characters
*/
        }
    }
    return fold ? (tolower(*s1) - tolower(*s2)) :
(*s1 - *s2);
}

/*numcmp:  compare s1 and s2 numerically */
int numcmp(char *s1, char *s2)
{
    double v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if(v1 < v2)
        return -1;
    else if(v1 > v2)
        return 1;
    else
        return 0;
}

void swap(void *v[], int i, int j)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

/*readlines:  read input lines.  This version is
slightly modified to read
  longer lines than the limit set by MAXLEN. */
int readlines(char *lineptr[], int maxlines)
```

```
{
    int len, nlines = 0;
    char *p, line[MAXLEN];
    int longline = 0;

    while((len = getline(line, MAXLEN)) > 0) {
        if(nlines >= maxlines || (p = alloc(len)) ==
NULL)
            return -1;
        else {
            if(line[len - 1] == '\n') {
                line[len - 1] = '\0';   /* delete
newline */
                strcpy(p, line);
                if(!longline)
                    lineptr[nlines++] = p;
                else
                    longline = 0;
            }
            else {
                strcpy(p, line);
                if(!longline) {
                    lineptr[nlines++] = p;
                    longline = 1;
                }
            }
        }
    }
    return nlines;
}

/* writelines: write output lines */
void writelines(char *lineptr[], int nlines)
{
    while(nlines-- > 0)
        printf("%s\n", *lineptr++);
    return;
}

int getline(char *s, int max)
{
```

```
    int c;
    char *ps = s;
    while(--max && (c=getchar()) != EOF && c != '\n')
        *s++ = c;
    if(c == '\n')
        *s++ = '\n';
    *s = '\0';
    return s - ps;
}

#define ALLOCSIZE 2000000

static char allocbuf[ALLOCSIZE];  /* storage for
alloc */
static char *allocp = allocbuf;    /* next free
position */

char *alloc(int n)
{
    if(allocbuf + ALLOCSIZE - allocp >= n) { /* it
fits */
        allocp += n;
        return allocp - n;          /* old p */
    }
    else                  /* not enough room */
        return 0;
}
```

17.

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

#define AUTHOR "Robert Taylor"
#define CREATION_DATE "May, 2014"
#define LAST_UPDATE __DATE__ /* last date that binary
was compiled */
static char *program_name;
```

```
/* My solution to Exercise 5-17
 * of the C Programming Language (second edition)
 * by Brian W. Kernighan
 * and Denis M. Ritchie
 *
 * To compile:
 * gcc -Os -Wall -s -o sort sort.c
 *
 * For help:
 * ./sort --h
 */

#define MAXLEN 1000 /* max length of any input line
*/
#define MAXLINES 500000 /* max # of lines to be
sorted */
#define ALLOCSIZE 15000000 /* size of space to store
lines */
#define NUMDIGITS 5 /* 4 digits plus the '\0'
terminator
              for input of numeric option values */
static char *lineptr[MAXLINES]; /* pointers to text
lines */

static int getLine(char *s, int lim);
static int readlines(char *lineptr[], int nlines);
static void writelines(char *lineptr[], int nlines);
static void my_qsort(void *lineptr[], int left, int
right, int (*comp)(const void*, const void*));
static int numcmp(const char *s1, const char *s2);
static int str_cmp(const char *s1, const char *s2);
static int compare(const char *s1, const char *s2);
static void swap(void *v[], int i, int j);
static int parse_args(int argc, char *argv[]);
static void recordinit(void); /* initialize each new
field record */
static void dump_field_records(void); /* dump field
records for analysis */
static void print_help(void); /* provide help on
program usage */
static void substr(char *s1, const char *s2, int p1,
```

```
int p2);
static void use_record(const char *s1, const char
*s2, int n);
static void substr_delim(char *s1, const char *s2,
int n);
static void dump_parsed_fields(char *lineptr[], int
nlines);
static void dump_parsed_fields_xml(char *lineptr[],
int nlines);

static int reverse = 0; /* 1 if sort in reverse order
*/
static int fold = 0; /* if fold = 1 it means case
insensitive sort */
static int directory = 0; /* if directory = 1 ignore
invalid characters */
static int numeric = 0; /* 1 if numeric sort */
static int num_lines_to_ignore = 0; /* ignore sorting
the first n lines */
static char line1[MAXLEN];/* two lines for
comparison, possibly substrings */
static char line2[MAXLEN];/* of the lines read */
static int sample_field_parse = 0; /* if 1, output
the data from the fields
                       defined, one field per line,
no sorting
                       is performed in this case*/
static int dump_as_xml = 0; /* when dumping parsed
fields, format as xml data */
static int include_line = 0; /* when dumping fields,
output source line
                 as well. */

/* logic to handle field records */
#define RECORDSIZE   9 /* number of values in the
record */
#define RECORDTYPE   0 /* offset within a record to
the record type */
#define FIELDSTART   1 /* start of field as offset
into the line */
#define FIELDEND 2 /* end of field as offset into the
```

```
line */
#define FIELDDELIM   1 /* also could store delim char
here */
#define FIELDQUOTE   2 /* also could store quote char
here */
#define FIELDNUMERIC 3 /* numeric sort for this
field? */
#define FIELDREV 4 /* sort this field in reverse
order? */
#define FIELDFOLD5 /* fold (case insensitive) sort
for this field? */
#define FIELDDIR 6 /* directory sort for this field?
*/
#define FIELDOFFSET  7 /* offset of field (counting
from left to right) */
#define FIELDESC 8 /* escape character typically \ */

#define RECORDTYPEOFFSET 0 /* 0 if we are considering
offsets into the line */
#define RECORDTYPEDELIM 1 /* 1 if we are considering
parsing delimited type data */

#define RECORDNUM 100 /* maximum number of field
records */

#define MAXFIELDS RECORDNUM * RECORDSIZE /* room to
hold RECORDNUM field records */
static int fieldinfo[MAXFIELDS]; /* array to hold
field record data */
static int numfields = 0; /* number of fields for
which options were read */
static int curfield = 0; /* field we are currently
considering */

/* variables used to carry over default/last-set
values between one field
 * record and the next */
static int recordtype = RECORDTYPEOFFSET; /* offsets
within lines */
static int field_delim = ','; /* delimiter between
fields */
```

```
static int field_quote = '"'; /* quote character */
static int field_esc = '\\'; /* for escaping
field_delim, field_quote
                     field_esc etc.*/

/* Sort input lines... sorts lines of input data
(STDIN) and sends sorted
 * lines to output (STDOUT). Run with --h to see
help.
 */

int main(int argc, char *argv[])
{
    int nlines; /* number of input lines read */
    recordinit();/* initialize the first field record
*/
    if (parse_args(argc, argv)) /* parse command line
arguments */
        return 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0)
{
        if (sample_field_parse){
            if (dump_as_xml){
                dump_parsed_fields_xml(lineptr,
nlines);
            } else {
                dump_parsed_fields(lineptr, nlines);
            }
        } else {
            my_qsort((void **) lineptr, 0, nlines
-1,
                     (int (*)(const void*,const
void*))compare);
            writelines(lineptr, nlines);
        }
        return 0;
    } else {
        printf("input too big to sort\n");
        return 1;
    }
}
```

```
/* parse_args: create field records from command line
arguments */
static int parse_args(int argc, char *argv[])
{
    int i = 0; /* index into each argument string */
    int j = 0; /* index into numstring */
    int no_room = 0; /* if 1 stop collecting field
records */
    int temp = 0; /* temporary storage of numbers */
    char numstring[NUMDIGITS];/* space to store
passed number as a string */
    program_name = &(*argv)[0];
    while (--argc > 0){
        if (no_room)
            break; /* stop collecting field records
*/
               ++argv; /* look at the next command
line argument */
        i = 0;
        j = 0;
        if (((*argv)[i] == ':') || ((*argv)[i] ==
',')){
            recordinit();
            if (numfields >= RECORDNUM)
                break; /* no room to store more
field records */
            continue;
        }
        if ((*argv)[i] == '-'){
            while ((*argv)[i] != '\0'){ /* allow for
mashed together options */
                if (no_room)
                    break; /* stop collecting field
records */
                i++;
                switch((*argv)[i]){
                case 'n':
                    numeric = 1;
                    fieldinfo[((numfields - 1) *
RECORDSIZE) + FIELDNUMERIC] = numeric;
                    break;
```

```
                 case 'r':
                      reverse = 1;
                      fieldinfo[((numfields - 1) *
RECORDSIZE) + FIELDREV] = reverse;
                      break;
                 case 'f':
                      fold = 1;
                      fieldinfo[((numfields - 1) *
RECORDSIZE) + FIELDFOLD] = fold;
                      break;
                 case 'd':
                      directory = 1;
                      fieldinfo[((numfields - 1) *
RECORDSIZE) + FIELDDIR] = directory;
                      break;
                 case 's':
                      i++;
                      j = 0;
                      while ((*argv)[i] != '\0' &&
isdigit((*argv)[i])){
                           numstring[j++] = (*argv)[i+
+];
                           if (j >= NUMDIGITS) /* do
not read too many digits */
                                break;
                      }
                      i--;
                      numstring[j] = '\0';
                      temp = atoi(numstring); /*
offset starts at 0 for first char */
                      if (temp >= MAXLEN){
                           printf("ERROR: The value
provided for the s option"
                                   " is too large\n");
                           return 1;
                      } else {
                           fieldinfo[((numfields - 1)
* RECORDSIZE) + FIELDSTART] = temp;
                      }
                      break;
                 case 'e':
```

```
                    i++;
                    j = 0;
                    /* slight alteration to support
passing a negative number */
                    while ((*argv)[i] != '\0' &&
(isdigit((*argv)[i]) || (*argv)[i] == '-')){
                        numstring[j++] = (*argv)[i+
+];
                        if (j >= NUMDIGITS) /* do
not read too many digits */
                            break;
                    }
                    i--;
                    numstring[j] = '\0';
                    temp = atoi(numstring);
                    if (temp >= MAXLEN){
                        printf("ERROR: The value
provided for the e option"
                                " is too large\n");
                        return 1;
                    } else {
                        fieldinfo[((numfields - 1)
* RECORDSIZE) + FIELDEND] = temp;
                    }
                    break;
                case 'o':
                    i++;
                    j = 0;
                    while ((*argv)[i] != '\0' &&
isdigit((*argv)[i])){
                        numstring[j++] = (*argv)[i+
+];
                        if (j >= NUMDIGITS) /* do
not read too many digits */
                            break;
                    }
                    i--;
                    numstring[j] = '\0';
                    temp = atoi(numstring);
                    if (temp >= MAXLEN){
                        printf("ERROR: The value
```

```
provided for the o option"
                                    " is too large\n");
                                return 1;
                            } else {
                                fieldinfo[((numfields - 1)
* RECORDSIZE) + FIELDOFFSET] = temp;
                            }
                            break;
                        case 't':
                            i++;
                            j = 0;
                            while ((*argv)[i] != '\0' &&
isdigit((*argv)[i])){
                                numstring[j++] = (*argv)[i+
+];
                                if (j >= NUMDIGITS) /* do
not read too many digits */
                                    break;
                            }
                            i--;
                            numstring[j] = '\0';
                            temp = atoi(numstring);
                            if (temp != RECORDTYPEOFFSET &&
temp != RECORDTYPEDELIM){
                                printf("ERROR: This is an
ivalid value for the"
                                    " t option\n");
                                return 1;
                            } else {
                                recordtype = temp;
                                fieldinfo[((numfields - 1)
* RECORDSIZE) + RECORDTYPE] = recordtype;
                                if (recordtype ==
RECORDTYPEDELIM){
                                    if
(fieldinfo[((numfields - 1) * RECORDSIZE) +
FIELDDELIM] == 0){

fieldinfo[((numfields - 1) * RECORDSIZE) +
FIELDDELIM] = field_delim;
                                    }
```

```
                                if
(fieldinfo[((numfields - 1) * RECORDSIZE) +
FIELDQUOTE] <= 0){

fieldinfo[((numfields - 1) * RECORDSIZE) +
FIELDQUOTE] = field_quote;
                                }
                                if
(fieldinfo[((numfields - 1) * RECORDSIZE) + FIELDESC]
== 0){

fieldinfo[((numfields - 1) * RECORDSIZE) + FIELDESC]
=    field_esc;
                                }
                            }
                        }
                        break;
                    case 'm':
                        i++;
                        if ((*argv)[i] == '1'){
                            i++;
                            if ((*argv)[i] != '\0'){
                                if ((*argv)[i] == 'S'
&& (*argv)[i + 1] == 'P'){
                                    field_delim = ' ';
                                    i++;
                                } else if ((*argv)[i]
== 'T' && (*argv)[i + 1] == 'A'){
                                    field_delim = '\t';
                                    i++;
                                } else if ((*argv)[i]
== 'V' && (*argv)[i + 1] == 'E'){
                                    field_delim = '|';
                                    i++;
                                } else if ((*argv)[i]
== 'S' && (*argv)[i + 1] == 'E'){
                                    field_delim = ';';
                                    i++;
                                } else if ((*argv)[i]
== 'B' && (*argv)[i + 1] == 'A'){
                                    field_delim = '\\';
```

```
                                        i++;
                                    } else if ((*argv)[i]
== 'P' && (*argv)[i + 1] == 'E'){
                                        field_delim = '%';
                                        i++;
                                    } else if ((*argv)[i]
== 'D' && (*argv)[i + 1] == 'O'){
                                        field_delim = '$';
                                        i++;
                                    } else if ((*argv)[i]
== 'S' && (*argv)[i + 1] == 'Q'){
                                        field_delim = '\'';
                                        i++;
                                    } else if ((*argv)[i]
== 'D' && (*argv)[i + 1] == 'Q'){
                                        field_delim = '"';
                                        i++;
                                    } else {
                                        field_delim =
(*argv)[i];
                                    }
                                    fieldinfo[((numfields -
1) * RECORDSIZE) + FIELDDELIM] = field_delim;
                                } else {
                                    printf("ERROR: Please
provide the field delimiter character"
                                            " immediately
following the -m1 option\n");
                                    return 1;
                                }
                            } else if ((*argv)[i] == '2'){
                                i++;
                                if ((*argv)[i] != '\0'){
                                    if ((*argv)[i] == 'S'
&& (*argv)[i + 1] == 'P'){
                                        field_quote = ' ';
                                        i++;
                                    } else if ((*argv)[i]
== 'T' && (*argv)[i + 1] == 'A'){
                                        field_quote = '\t';
                                        i++;
```

```
                                } else if ((*argv)[i]
== 'V' && (*argv)[i + 1] == 'E'){
                                    field_quote = '|';
                                    i++;
                                } else if ((*argv)[i]
== 'S' && (*argv)[i + 1] == 'E'){
                                    field_quote = ';';
                                    i++;
                                } else if ((*argv)[i]
== 'B' && (*argv)[i + 1] == 'A'){
                                    field_quote = '\\';
                                    i++;
                                } else if ((*argv)[i]
== 'P' && (*argv)[i + 1] == 'E'){
                                    field_quote = '%';
                                    i++;
                                } else if ((*argv)[i]
== 'D' && (*argv)[i + 1] == 'O'){
                                    field_quote = '$';
                                    i++;
                                } else if ((*argv)[i]
== 'S' && (*argv)[i + 1] == 'Q'){
                                    field_quote = '\'';
                                    i++;
                                } else if ((*argv)[i]
== 'D' && (*argv)[i + 1] == 'Q'){
                                    field_quote = '"';
                                    i++;
                                } else if ((*argv)[i]
== 'N' && (*argv)[i + 1] == 'U'){
                                    field_quote = '\0';
                                    i++;
                                } else {
                                    field_quote =
(*argv)[i];
                                }
                                fieldinfo[((numfields -
1) * RECORDSIZE) + FIELDQUOTE] = field_quote;
                            } else {
                                printf("ERROR: Please
provide the field quote character"
```

```
                                              " immediately
following the -m2 option\n");
                              return 1;
                          }
                      } else if ((*argv)[i] == '3'){
                          i++;
                          if ((*argv)[i] != '\0'){
                              if ((*argv)[i] == 'S'
&& (*argv)[i + 1] == 'P'){
                                  field_esc = ' ';
                                  i++;
                              } else if ((*argv)[i]
== 'T' && (*argv)[i + 1] == 'A'){
                                  field_esc = '\t';
                                  i++;
                              } else if ((*argv)[i]
== 'V' && (*argv)[i + 1] == 'E'){
                                  field_esc = '|';
                                  i++;
                              } else if ((*argv)[i]
== 'S' && (*argv)[i + 1] == 'E'){
                                  field_esc = ';';
                                  i++;
                              } else if ((*argv)[i]
== 'B' && (*argv)[i + 1] == 'A'){
                                  field_esc = '\\';
                                  i++;
                              } else if ((*argv)[i]
== 'P' && (*argv)[i + 1] == 'E'){
                                  field_esc = '%';
                                  i++;
                              } else if ((*argv)[i]
== 'D' && (*argv)[i + 1] == 'O'){
                                  field_esc = '$';
                                  i++;
                              } else if ((*argv)[i]
== 'S' && (*argv)[i + 1] == 'Q'){
                                  field_esc = '\'';
                                  i++;
                              } else if ((*argv)[i]
== 'D' && (*argv)[i + 1] == 'Q'){
```

```
                                                field_esc = '"';
                                                i++;
                                        } else if ((*argv)[i]
== 'N' && (*argv)[i + 1] == 'U'){
                                                field_esc = '\0';
                                                i++;
                                        } else {
                                                field_esc = (*argv)
[i];
                                        }
                                        fieldinfo[((numfields -
1) * RECORDSIZE) + FIELDESC] = field_esc;
                                } else {
                                        printf("ERROR: Please
provide the desired escape character"
                                                " immediately
following the -m3 option\n");
                                        return 1;
                                }
                        }
                        break;
                case 'i':
                        i++;
                        j = 0;
                        while ((*argv)[i] != '\0' &&
isdigit((*argv)[i])){
                                numstring[j++] = (*argv)[i+
+];
                                if (j >= NUMDIGITS) /* do
not read too many digits */
                                        break;
                        }
                        i--;
                        numstring[j] = '\0';
                        num_lines_to_ignore =
atoi(numstring);
                        break;
                case '-':
                        i++;
                        /* --help or --HELP or --h or --
H */
```

```
                    if ((*argv)[i] == 'h' || (*argv)
[i] == 'H'){
                        print_help();
                        return 1; /* exit */
                    }
                    /* --dump_records or --
DUMP_RECORDS or --d or --D etc. */
                    if ((*argv)[i] == 'd' || (*argv)
[i] == 'D'){
                        dump_field_records();
                        return 1; /* exit */
                    }
                    /* --sample_field_parse or --
SAMPLE_FIELD_PARSE or --s or --S */
                    if ((*argv)[i] == 's' || (*argv)
[i] == 'S'){
                        sample_field_parse = 1;
                        break; /* in case we add
more ifs below */
                    }
                    /* --xml or --XML or --x or --X
*/
                    if ((*argv)[i] == 'x' || (*argv)
[i] == 'X'){
                        dump_as_xml = 1;
                        break; /* in case we add
more ifs below */
                    }
                    /* --include_line or --
INCLUDE_LINE or --i or --I */
                    if ((*argv)[i] == 'i' || (*argv)
[i] == 'I'){
                        include_line = 1;
                        break; /* in case we add
more ifs below */
                    }
                    break;
                case ':':
                    recordinit();
                    if (numfields >= RECORDNUM){
                        no_room = 1; /* no room to
```

```
store more field records */
                    }
                    break;
               case ',':
                    recordinit();
                    if (numfields >= RECORDNUM){
                         no_room = 1; /* no room to
store more field records */
                    }
                    break;
               default:
                    break;
               }
          }
     }
  }
  return 0;
}

/* Initialize some values for each new field record
added */
static void recordinit(void)
{
     fieldinfo[(numfields * RECORDSIZE) + RECORDTYPE]
= recordtype;
     if (recordtype){
          fieldinfo[(numfields * RECORDSIZE) +
FIELDDELIM] = field_delim;
          fieldinfo[(numfields * RECORDSIZE) +
FIELDQUOTE] = field_quote;
     } else {
          fieldinfo[(numfields * RECORDSIZE) +
FIELDSTART] = 0;
          fieldinfo[(numfields * RECORDSIZE) +
FIELDEND] = -1;
     }
     fieldinfo[(numfields * RECORDSIZE) +
FIELDNUMERIC] = 0;
     fieldinfo[(numfields * RECORDSIZE) + FIELDREV] =
0;
     fieldinfo[(numfields * RECORDSIZE) + FIELDFOLD] =
```

```
0;
    fieldinfo[(numfields * RECORDSIZE) + FIELDDIR] =
0;
    fieldinfo[(numfields * RECORDSIZE) + FIELDOFFSET]
= 0;
    fieldinfo[(numfields * RECORDSIZE) + FIELDESC] =
field_esc;
    numfields++;
    return;
}

/* use_record: Setup to use the values from the
specified
 * field record.
 */
static void use_record(const char *s1, const char
*s2, int n)
{
    int fieldnum, p1, p2;
    /* set some sourcefile scoped global variables */
    recordtype = fieldinfo[(n * RECORDSIZE) +
RECORDTYPE];
    numeric = fieldinfo[(n * RECORDSIZE) +
FIELDNUMERIC];
    reverse = fieldinfo[(n * RECORDSIZE) + FIELDREV];
    fold = fieldinfo[(n * RECORDSIZE) + FIELDFOLD];
    directory = fieldinfo[(n * RECORDSIZE) +
FIELDDIR];
    /* set type specific variables and initialize
line1 and line2 */
    if (recordtype){
        field_delim = fieldinfo[(n * RECORDSIZE) +
FIELDDELIM];
        field_quote = fieldinfo[(n * RECORDSIZE) +
FIELDQUOTE];
        fieldnum = fieldinfo[(n * RECORDSIZE) +
FIELDOFFSET];
        field_esc = fieldinfo[(n * RECORDSIZE) +
FIELDESC];
        substr_delim(line1, s1, fieldnum);
        substr_delim(line2, s2, fieldnum);
```

```
    } else {
        p1 = fieldinfo[(n * RECORDSIZE) +
FIELDSTART];
        p2 = fieldinfo[(n * RECORDSIZE) + FIELDEND];
        substr(line1, s1, p1, p2);
        substr(line2, s2, p1, p2);
    }
    //printf("line1 is %s line2 is %s\n", line1,
line2);
    return;
}

/* my_qsort: sort v[left] ... v[right] into
 * increasing or decreasing order depending on
 * the value of reverse.*/
static void my_qsort(void *v[], int left, int right,
        int (*comp)(const void*, const void*))
{
    int i, last;
    if (left >= right) /* do nothing if array
contains */
        return; /* fewer than two elements */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left + 1; i <= right; i++)
        if ((*comp)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    my_qsort(v, left, last - 1, comp);
    my_qsort(v, last + 1, right, comp);
}
/* compare: Parent process for comparisons. I am able
 * to dynamically make decisions concerning whether
numcmp
 * or str_cmp should be called in here.
 * Whether you code the reverse logic here or in
qsort, it
 * needs to support changing dynamically based on
what field
 * record is currently loaded.
 */
```

```c
static int compare(const char *s1, const char *s2)
{
    int retval;
    curfield = 0;
    do{
        use_record(s1, s2, curfield);
        if (numeric)
            retval = numcmp(line1, line2);
        else
            retval = str_cmp(line1, line2);
        if (retval == 0){
            curfield++;
        }
    } while (retval == 0 && curfield < numfields);
    if (reverse){
        if (retval > 0){
            retval = -1;
        } else if (retval < 0){
            retval = 1;
        }
    }
    return retval;
}

/* numcmp: compare s1 and s2 numerically */
static int numcmp(const char *s1, const char *s2)
{
    double v1, v2;
    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
/* str_cmp: replaces standard library strcmp to add
 * more features for types of comparison. Supports
 * case insensitive comparison for example. Supports
 * ignoring invalid characters. Borrowed version of
```

```
this
 * function by Barrett Drawdy as it was cleaner than
 * mine.
 */
static int str_cmp(const char *s1, const char *s2)
{
    if (directory){
        while (!isdigit(*s1) && !isalpha(*s1) && !
isspace(*s1) && *s1)
            ++s1; /* ignore bad characters */
        while (!isdigit(*s2) && !isalpha(*s2) && !
isspace(*s2) && *s2)
            ++s2; /* ignore bad characters */
    }
    while (fold ? (tolower(*s1) == tolower(*s2)) :
(*s1 == *s2)){
        if (*s1 == '\0')
            return 0;
        ++s1;
        ++s2;
        if (directory){
            while (!isdigit(*s1) && !isalpha(*s1)
&& !isspace(*s1) && *s1)
                ++s1; /* ignore bad characters */
            while (!isdigit(*s2) && !isalpha(*s2)
&& !isspace(*s2) && *s2)
                ++s2; /* ignore bad characters */
        }
    }
    return fold ? (tolower(*s1) - tolower(*s2)) :
(*s1 - *s2);
}
/* copy characters from offset p1 to p2 (inclusive)
of s2 to s1 */
static void substr(char *s1, const char *s2, int p1,
int p2)
{
    int i, j;
    int length = strlen(s2);
    if (p1 + 1 >= length){ /* desired field is
missing from line */
```

```
        s1[0] = '\0';
        return;
    }
    if (p2 < p1)
        p2 = length;
    /* if p2 is too big we will simply set s1 to
whatever is left */
    for (i = p1, j = 0; i <= p2 && s2[i] != '\0'; i+
+, j++)
        s1[j] = s2[i];
    s1[j] = '\0';
    return;
}

/* substr_delim: copy characters from the delimited
field number n of
 * s2 to s1.
 * field_delim is counted if it is outside of a pair
of field_quote
 * (a quoted field), and not escaped with a
field_esc. Otherwise
 * it is just copied as part of the data.
 *
 * If field_esc is followed by field_esc,
field_delim, or field_quote
 * it is skipped and the following character is
copied as part of
 * the data. Otherwise field_esc is just copied as
part of the data.
 *
 * field_quote only has special meaning if it is at
the very beginning
 * of a field, or if it is at the very end of a field
and follows a
 * field_quote that was at the very beginning of a
field. If field_quote
 * is seen in the middle of the field it is just
considered part of the
 * data.
 *
 * This logic should allow substr_delim to work with
```

```
data that has been
 * prepared using strict quoting and escaping rules,
while allowing
 * the most flexibility for handling data that was
not strictly quoted
 * and escaped.
 *
 */
static void substr_delim(char *s1, const char *s2,
int n)
{
    int delim_count = 0; /* the first field is number
0 */
    int field_quote_on = 0; /* are we inside of
quotes? */
    /* i, j,  indexes into s2 and s1 */
    int i = 0;
    int j = 0;
    while (s2[i] != '\0'){
        if (i == 0 && s2[i] == field_quote){
            field_quote_on = 1;
            ++i;
            continue;
        }
        if (s2[i] == field_esc){
            if (s2[i + 1] == field_quote ||
                    s2[i + 1] == field_delim ||
                    s2[i + 1] == field_esc){
                ++i; /* skip this field_esc and copy
the next char */
                        s1[j++] = s2[i++];
                        continue;
            }
        }
        if (field_quote_on == 1 &&
                s2[i] == field_quote &&
                (s2[i + 1] == field_delim || s2[i +
1] == '\0')){
            field_quote_on = 0;
            ++i;
            continue;
```

```
        }
        if (field_quote_on == 0 && s2[i] ==
field_delim){
            ++delim_count;
            ++i;
            if (s2[i] == field_quote){
                field_quote_on = 1;
                ++i;
            }
            continue;
        }
        if (delim_count == n){
            s1[j++] = s2[i];
        } else if (delim_count > n){
            break;
        }
        ++i;
    }
    s1[j] = '\0';
    return;
}

/* swap pointers: void * is used so that swap can
work on any pointer type.
 * Any pointer can be cast to void * and back again
without loss of
 * information
 */
static void swap(void *v[], int i, int j)
{
    void *temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

static char *alloc(int);
/* readlines: read input lines */
static int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
```

```
    char *p, line[MAXLEN];
    nlines = 0;
    while ((len = getLine(line, MAXLEN)) > 0)
        if (num_lines_to_ignore){
            if (!sample_field_parse) /* if we are
sampling fields we may not
                              want the ignored lines
output */
                printf("%s", line);
            --num_lines_to_ignore;
        } else {
            if(nlines >= maxlines || (p =
alloc(len)) == NULL)
                return -1;
            else {
                line[len - 1] = '\0'; /* delete
newline */
                strcpy(p, line);
                lineptr[nlines++] = p;
            }
        }
    return nlines;
}

/* writelines: write output lines */
static void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}

/* getLine: read a line into s, return length */
static int getLine(char *s, int lim)
{
    int c, i;
    for (i = 0; i < lim - 1 && (c = getchar()) != EOF
&& c != '\n'; ++i)
        s[i] = c;
    if (c == '\n'){
        s[i] = c;
        ++i;
```

```
    }
    s[i] = '\0';
    return i;
}

static char allocbuf[ALLOCSIZE]; /* storage for alloc
*/
static char *allocp = allocbuf; /* next free position
*/
static char *alloc(int n) /* return pointer to n
characters */
{
    if (allocbuf + ALLOCSIZE - allocp >= n){ /* it
fits */
        allocp += n;
        return allocp - n; /* old p */
    } else {
        return NULL;
    }
}


/* dump_parsed_fields: Apply the field position/
offset/delimiters
 * defined and dump the resulting field contents, one
field per
 * line, repeat for all input lines. Useful for
debugging how
 * the field position and contents are identified for
the sort
 */
static void dump_parsed_fields(char *lineptr[], int
nlines)
{
    int n = 0;
    int i = 1;
    int fieldnum, p1, p2;
    while (i <= nlines){
        if (include_line)
            printf("%s\n", *lineptr);
        n = 0;
```

```
        do{
            recordtype = fieldinfo[(n * RECORDSIZE)
+ RECORDTYPE];
            if (recordtype){
                field_delim = fieldinfo[(n *
RECORDSIZE) + FIELDDELIM];
                field_quote = fieldinfo[(n *
RECORDSIZE) + FIELDQUOTE];
                fieldnum = fieldinfo[(n *
RECORDSIZE) + FIELDOFFSET];
                field_esc = fieldinfo[(n *
RECORDSIZE) + FIELDESC];
                substr_delim(line1, *lineptr,
fieldnum);
            } else {
                p1 = fieldinfo[(n * RECORDSIZE) +
FIELDSTART];
                p2 = fieldinfo[(n * RECORDSIZE) +
FIELDEND];
                substr(line1, *lineptr, p1, p2);
            }
            printf("%s\n", line1);/* output the
parsed field */
            n++;
        } while (n < numfields);
        lineptr++;
        i++;
    }
    return;
}
/* dump_parsed_fields_xml: Apply the field position/
offset/delimiters
 * defined and dump the resulting field contents, in
an xml format.
 */
static void dump_parsed_fields_xml(char *lineptr[],
int nlines)
{
    int n = 0;
    int i = 1;
    int fieldnum, p1, p2;
```

```
    printf("<root>\n");
    while (i <= nlines){
        printf("\t<line%d>", i);
        n = 0;
        do{
            recordtype = fieldinfo[(n * RECORDSIZE)
+ RECORDTYPE];
            if (recordtype){
                field_delim = fieldinfo[(n *
RECORDSIZE) + FIELDDELIM];
                field_quote = fieldinfo[(n *
RECORDSIZE) + FIELDQUOTE];
                fieldnum = fieldinfo[(n *
RECORDSIZE) + FIELDOFFSET];
                field_esc = fieldinfo[(n *
RECORDSIZE) + FIELDESC];
                substr_delim(line1, *lineptr,
fieldnum);
            } else {
                p1 = fieldinfo[(n * RECORDSIZE) +
FIELDSTART];
                p2 = fieldinfo[(n * RECORDSIZE) +
FIELDEND];
                substr(line1, *lineptr, p1, p2);
            }
            printf("<f%d>%s</f%d>", n, line1, n);/*
output the parsed field */
            n++;
        } while (n < numfields);
        printf("</line%d>\n", i);
        lineptr++;
        i++;
    }
    printf("</root>\n");
    return;
}

/* dump_field_records: for debugging purposes you may
wish to
 * dump all the field records that have been stored
 */
```

```
static void dump_field_records(void)
{
    int i;
    int c;
    printf("\n---------------------------| Field
Record Dump |-----------------------------\n");
    printf("Record    | COL0    | COL1    | COL2    | COL3
    | COL4    | COL5    | COL6    | COL7    | COL8\n");
    for (i = 0; i < numfields; i++){
        printf(" %d\t", i);
        printf("| %d\t", fieldinfo[(i * RECORDSIZE) +
RECORDTYPE]);
        if (fieldinfo[(i * RECORDSIZE) + RECORDTYPE]
== RECORDTYPEOFFSET){
            printf("| %d\t", fieldinfo[(i *
RECORDSIZE) + FIELDSTART]);
            printf("| %d\t", fieldinfo[(i *
RECORDSIZE) + FIELDEND]);
        } else {
            c = fieldinfo[(i * RECORDSIZE) +
FIELDDELIM];
            if (c == '\t'){
                printf("| TAB\t");
            } else if (c == ' '){
                printf("| SPACE\t");
            } else {
                printf("| %c\t", c);
            }
            c = fieldinfo[(i * RECORDSIZE) +
FIELDQUOTE];
            if (c == '\t'){
                printf("| TAB\t");
            } else if (c == ' '){
                printf("| SPACE\t");
            } else if (c == '\0'){
                printf("| NULL\t");
            } else {
                printf("| %c\t", c);
            }
        }
        printf("| %d\t",fieldinfo[(i * RECORDSIZE) +
```

```
FIELDNUMERIC]);
        printf("| %d\t",fieldinfo[(i * RECORDSIZE) +
FIELDREV]);
        printf("| %d\t",fieldinfo[(i * RECORDSIZE) +
FIELDFOLD]);
        printf("| %d\t",fieldinfo[(i * RECORDSIZE) +
FIELDDIR]);
        printf("| %d\t",fieldinfo[(i * RECORDSIZE) +
FIELDOFFSET]);
        c = fieldinfo[(i * RECORDSIZE) + FIELDESC];
        if (c == '\t'){
            printf("| TAB\t");
        } else if (c == ' '){
            printf("| SPACE\t");
        } else if (c == '\0'){
            printf("| NULL\t");
        } else {
            printf("| %c\t", c);
        }
    }
    printf("\nWhere COL0 = RECORDTYPE, COL1 =
FIELDSTART or FIELDDELIM\n");
    printf("COL2 = FIELDEND or FIELDQUOTE, COL3 =
FIELDNUMERIC, COL4 = FIELDREV\n");
    printf("COL5 = FIELDFOLD, COL6 = FIELDDIR, COL7 =
FIELDOFFSET, COL8 = FIELDESC\n\n");
    printf("RECORDTYPE is set by -t\n");
    printf("FIELDSTART is set by -s\n");
    printf("FIELDDELIM is set by -m1\n");
    printf("FIELDEND is set by -e\n");
    printf("FIELDQUOTE is set by -m2\n");
    printf("FIELDNUMERIC is set by -n\n");
    printf("FIELDREV is set by -r\n");
    printf("FIELDFOLD is set by -f\n");
    printf("FIELDDIR is set by -d\n");
    printf("FIELDOFFSET is set by -o\n");
    printf("FIELDESC is set by -m3\n");
    printf("For more info use the --h (help) option.
\n\n");
    return;
}
```

```
static void print_help(void)
{
    printf("Program: %s\n", program_name);
    printf("Author: %s\n", AUTHOR);
    printf("Creation Date: %s\n", CREATION_DATE);
    printf("Last Update: %s\n", LAST_UPDATE);
    printf("usage: cat sourcefile | ./sort -options
[> outputfile]\n\n");
    printf("This sort program expects input from
STDIN (output from cat for\n");
    printf("example) and sends output to STDOUT (the
screen for example).\n\n");
    printf("If you fail to provide input from cat or
similar you will be in\n");
    printf("an interactive input mode. This means you
can enter lines using\n");
    printf("the keyboard and press CTRL-D on an empty
line to process them.\n\n");
    printf("Pressing CTRL-C will abort the program.\n
\n");
    printf("%s is quite sophisticated, permitting you
to break a line\n", program_name);
    printf("into fields that can have separate
sorting options applied to\n");
    printf("them.\n\n");
    printf("It is always lines that are sorted, not
the fields within the line.\n");
    printf("However defining fields and specifying
sort options for them permits\n");
    printf("sophisticated sorting behavior for the
lines.\n\n");
    printf("You do not need to provide sorting
options for every field on a line.\n");
    printf("Only specify those fields that have the
data that you wish to use\n");
    printf("to sort the lines.\n\n");
    printf("The order that you specify the fields on
the command line is the order\n");
    printf("of precedence for sorting.\n\n");
    printf("For example...\n");
```

```
    printf("if one field has a username and another has a date, you can sort\n");
    printf("by increasing (ascending) username and then by decreasing (descending)\n");
    printf("date simply by specifying those 2 fields and their sort options on the\n");
    printf("command line. Specify the username field first and the date field\n");
    printf("next. The way the logic works is only if there are equal values\n");
    printf("found in the first field, is the second field examined, and only if\n");
    printf("there are equal values found in the second field is the third field\n");
    printf("examined, and so on until either a difference is found between the\n");
    printf("two fields in question, or we run out of fields that we have defined\n");
    printf("for sorting. If we run out of fields that we have defined for sorting\n");
    printf("and no difference has been found, the lines are considered equal.\n\n");
    printf("This sort program supports the use of up to %d fields with unique\n", RECORDNUM);
    printf("sort options permitted for each field. The data can have any number of\n");
    printf("fields, but you can only specify sort options for %d fields.\n", RECORDNUM);
    printf("Actually the data is limited in fields per line by the setting\n");
    printf("for the maximum line length of %d characters and the size of the fields.\n\n",MAXLEN);
    printf("The field definitions (size/location) along with the sort options\n");
    printf("are saved internally in records. There is a handy option that\n");
    printf("permits dumping these internal records so that you can evaluate\n");
    printf("how the program has interpreted the command line options that you\n");
    printf("have provided. Place this option --d (--
```

```
dump_records) after all\n");
    printf("the options that you desire to audit have
been specified on the command\n");
    printf("line.\n\n");
    printf("--d  dump field records, place after
other options on the command line.\n");
    printf(" If you use this option the sort is not
performed, this option is used\n");
    printf(" strictly for debugging your sort options
that you have defined\n\n");
    printf("%s allows you to specify fields using
character offsets, from the\n", program_name);
    printf("start of the line at position 0, or using
delimiters and quote characters.\n");
    printf("Each field definition can optionally use
either method.\n");
    printf("-t#  type of field definition, where #
is %d for the character offset method\n",
RECORDTYPEOFFSET);
    printf(" and %d for the delimited field method.
This option should be listed\n", RECORDTYPEDELIM);
    printf(" first in the field definition, but
inherits from left to right so\n");
    printf(" if all field definitions are of the same
type it only needs to be\n");
    printf(" specified for the first field for
example. The default is the\n");
    printf(" character offset method\n");
    printf("-s#  starting character position, where #
is a number that is less than\n");
    printf(" the maximum line length of %d\n",
MAXLEN);
    printf("-e#  ending character position, where #
is a number that is less than\n");
    printf(" the maximum line length of %d\n",
MAXLEN);
    printf("If you choose to use the character offset
method of defining a field you must\n");
    printf("set -s and -e for each field to the
correct offsets, sort will not check to\n");
    printf("see that you did.\n\n");
```

```
    printf("If you set -e to less than -s, it means
that you want from -s to the end of\n");
    printf("the line. The default setting for sort is
to use the character offset method\n");
    printf("and -s is set to 0 (the beginning) and -e
is set to -1 (the rest of the line).\n");
    printf("Because of the special meaning of -e set
to be less than -s, -e supports\n");
    printf("passing a negative number -e-1 for -1.\n
\n");
    printf("The default delimiters for delimited data
is to use a comma ',' to separate\n");
    printf("fields and to use double quotes '\"', to
quote fields. The quotes surround the\n");
    printf("fields to indicate that any commas that
are found within the fields can be\n");
    printf("ignored.\n\n");
    printf("In order to support the possibility of a
double quote found within the data\n");
    printf("an escape character can be used and the
default escape character is a\n");
    printf("backslash '\\'.\n\n");
    printf("In actuality, this program is coded to be
more flexible than that. An escape\n");
    printf("character 'can' be used to escape the
delimiter, the quote character or an\n");
    printf("escape character, either inside or
outside of a quoted field. If the escape\n");
    printf("character is found in front of anything
else it is considered part of the data.\n\n");
    printf("If the quote character is found anywhere
besides the start or end of a field it\n");
    printf("is considered part of the data and as
such, technically, does not need to be\n");
    printf("escaped. So this program should be able
to handle data formatted according to a\n");
    printf("variety of specifications.\n\n");
    printf("If this is not enough, the quote and/or
the escape character can also be\n");
    printf("disabled in cases where they are not
required and yet may be found in the data\n");
```

```
    printf("(see the NU code below).\n\n");
    printf("You can define what characters to use to
separate fields, to quote fields,\n");
    printf("and to escape quote characters.\n\n");
    printf("-m1n field separator, where n is the
desired character\n");
    printf("-m2n quote character, where n is the
desired character\n");
    printf("-m3n escape character, where n is the
desired escape character\n\n");
    printf("In the above 3 options instead of
specifying the literal character desired\n");
    printf("as n you can use the following 2 letter
(uppercase) codes:\n\n");
    printf("SP   to mean a SPACE\n");
    printf("TA   to mean a TAB\n");
    printf("VE   to mean a VERTICAL BAR '|'\n");
    printf("SE   to mean a SEMICOLON ';'\n");
    printf("BA   to mean a BACKSLASH '\\'\n");
    printf("PE   to mean a PERCENT SIGN '%%'\n");
    printf("DO   to mean a DOLLAR SIGN '$'\n");
    printf("SQ   to mean a SINGLE QUOTE '\n");
    printf("DQ   to mean a DOUBLE QUOTE \"\n\n");
    printf("NU   to mean NULL '\\0', is supported for
the quote or escape character.\n");
    printf(" Since such a character will not be seen
in the data (it is the string\n");
    printf(" terminator) it is used to disable the
operation of the quote or escape\n");
    printf(" character if that is ever desired.\n
\n");
    printf("When using the delimited field method to
specify fields it is important to\n");
    printf("indicate which field in the data we are
referring to. Counting from 0 for the\n");
    printf("leftmost field on a line you can indicate
field numbers using the -o option.\n");
    printf("-o#  indicate which delimited field,
where # is a number from 0 to however\n");
    printf(" many fields exist in the data\n\n");
    printf("Every new field record initializes -o to
```

```
be 0, referring to the first field, if\n");
    printf("this is not the field you want you must
set the -o option to the correct field\n");
    printf("number.\n\n");
    printf("Using either method of specifying a
field, the character offset method or the\n");
    printf("delimited method, if a specified field
does not exist in the line in question\n");
    printf("it is treated as an empty field. If a
large number of lines are missing this\n");
    printf("field and it is the only sort field that
you indicated the sort will be slow.\n");
    printf("Qsort does not like it when too many
lines evaluate to be equal.\n\n");
    printf("Between field definitions, to indicate
the start of a new field, you can use\n");
    printf("either a colon ':' or a comma ','. Be
careful not to place a field separator at\n");
    printf("the beginning as by default it will
indicate to use the entire line with the\n");
    printf("default sorting options as the first
field for sorting.\n\n");
    printf("The sorting options are:\n\n");
    printf("-n   numeric sort, puts numbers in order
of value\n");
    printf("-r   reverse the sort order, instead of
increasing order it would be\n");
    printf(" decreasing order\n");
    printf("-f   fold upper and lower case together,
or in other words do a case\n");
    printf(" insensitive sort\n");
    printf("-d   directory sort, this ignores any
character that is not a letter,\n");
    printf(" number or space\n\n");
    printf("The default sort options are set to have
all of these options off, which\n");
    printf("means punctuation characters, or other
special characters have a sorting\n");
    printf("value, the sort is in increasing order,
upper and lower case letters\n");
    printf("have different sorting values, 17 would
```

```
be considered lower than 2\n");
    printf("(numeric value is not considered).\n\n");
    printf("Options can be specified individually on
the command-line separated by\n");
    printf("spaces...\n");
    printf("cat sourcefile | ./sort -t1 -o1 -n , -o0
-f -d\n\n");
    printf("or they can be mashed together...\n");
    printf("cat sourcefile | ./sort -t1o1n,o0fd\n
\n");
    printf("However do not put spaces between an
option and its value. -s 99 is not\n");
    printf("an accepted parameter, the option should
be indicated as -s99. Also the\n");
    printf("use of an equals sign '=' is not
supported between an option and its\n");
    printf("value.\n\n");
    printf("Do not confuse the use of the -s (start)
and -e (end) options together\n");
    printf("with the delimited field option. It will
replace whatever is being used\n");
    printf("for the delimiter and quote character
with whatever character happens to\n");
    printf("equal the numeric value you provide. If
you really know what you are\n");
    printf("doing it can be useful, otherwise avoid
it.\n\n");
    printf("There is an option to ignore the first x
number of lines. You might use\n");
    printf("this option if your data includes a
header line with column titles for\n");
    printf("example and you do not want this line
sorted in with the data.\n");
    printf("-i#  ignore the first # of lines, where #
is a number between 0 and 9999.\n");
    printf(" 0 in this case has no logical meaning
since we start counting lines\n");
    printf(" with 1 as the first line.\n\n");
    printf("A Sorting Example:\n");
    printf("cat sourcefile | ./sort -t1o1n,o0fd\n
\n");
```

```
    printf("Means use the delimited field method (-
t1), the primary sort field is the\n");
    printf("second field (o1), do a numerical sort on
that field (n), if two identical\n");
    printf("numbers are found in the second field,
look at the first field (o0), doing a\n");
    printf("case insensitive directory sort of the
first field, decide the order for\n");
    printf("the two lines. The default delimiter,
quote character, and escape character\n");
    printf("is used.\n\n");
    printf("If the sort appears to be excessively
slow, it could be because the fields and\n");
    printf("options that you have selected result in
too many lines that would have an\n");
    printf("equal sort order. I think this is a
limitation of the qsort logic since\n");
    printf("specifying more fields and more options,
so that the lines which evaluate\n");
    printf("to be equal are reduced, significantly
speeds up the sort.\n\n");
    printf("A few additional options have been added
to aid in debugging the\n");
    printf("specification of fields. You may find
other uses for them as well. The option\n\n");
    printf("--s  for sample field parsing\n\n");
    printf("found anywhere in the options will
signify that instead of sorting the input\n");
    printf("you want to output the data from the
fields that you have indicated. This\n");
    printf("will assist you in debugging delimiters
and offset etc. when specifying\n");
    printf("fields for delimited data, and
identifying errors in character offset\n");
    printf("specification when using the character
offset method of identifying fields.\n");
    printf("It can be most useful when analyzing one
field specifier at a time as the\n");
    printf("fields will be dumped one field per line
so as not to introduce new field\n");
    printf("separators to confuse the issue when
```

```
identifying what data was exactly\n");
    printf("pulled.\n\n");
    printf("There is a further specifier that can be
added to also dump the source line\n");
    printf("that the fields were pulled from\n\n");
    printf("--i  for include line with the field
parse dump\n\n");
    printf("For assisting with debugging what is
parsed when multiple fields are specified\n");
    printf("there is an option to format the fields
into an xml document. No effort is made\n");
    printf("to escape invalid characters found in the
data, the data is left as is.\n\n");
    printf("--x  format parsed fields into an xml
document format\n\n");
    printf("If you specify to use the xml format, the
--i option is ignored, there is no\n");
    printf("option to include the original line in
the xml format. The field numbering\n");
    printf("used in the xml output is not the number
of the field with respect to the\n");
    printf("data but the number of the field record
specified on the command line, staring\n");
    printf("with 0 as the leftmost field record
specified\n\n");
    printf("Unless --s is specified --x and --i will
be ignored.\n\n");
    return;
}
```

18.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "getch.h"

#define MAXTOKEN 100
/* Author: Robert Taylor
 * Creation Date: June, 2014
 * Exercise 5-18
```

```
 * "Make dcl recover from input errors"
 * If you are doing this exercise for a course, you
probably would
 * not be expected to do as much as you will find in
this source file.
 *
 * You should be able to copy and paste most
declarations from a
 * source file that you might find to the input to
this dcl program.
 *
 * Notable limitations include: the reference
operator '&' is not
 * handled, a symbolic constant that is in an array
subscript
 * will be seen as an invalid subscript, type
modifiers/qualifiers such as
 * static, const, long, short, unsigned, signed etc.
are not handled as they
 * will be part of the answer for Exercise 5-20.
 *
 * This dcl can handle multiple declarations on a
line such as
 * char *p, line[MAXLEN];
 * dcl can also handle comments, and detects end of
declaration indicators
 * such as ; or { or =
 *
 * Recovery from errors is interpreted to mean that
despite missing one of
 * [ or ] or ( or ) or NAME, an error message should
be displayed and some
 * educated interpretation should be attempted. In
the case of parentheses
 * it is sometimes difficult to know where they
should have been so the
 * resulting interpretation will not necessarily be
correct.
 *
 * In the case of a missing NAME, instead of
outputting an error the name
```

```
 * is simply omitted from the output. The following
for example:
 * char (*(*())[])()
 * will output:
 * :  function returning pointer to array[] of
pointer to function returning char
 */

enum { NAME, PARENS, BRACKETS };

static void dcl(void);
static void dirdcl(void);

static int gettoken(void);
static int tokentype; /* type of last token */
static int oldtoken; /* previous tokentype */
static char token[MAXTOKEN]; /* last token string */
static char name[MAXTOKEN]; /* identifier name */
static char datatype[MAXTOKEN]; /* data type = char,
int, etc. */
static char out[1000]; /* output string */
static int parenopen = 0; /* count of open
parentheses */
static int parenclose = 0; /* count of unmatched
close parentheses */
static int alphaseen = 0; /* track alpha statis for
subscript */

int main(void) /* convert declaration to words */
{
    while (gettoken() != EOF) { /* 1st token on line
*/
        strcpy(datatype, token); /* pull data type */
        while (tokentype != '\n'){
            out[0] = '\0';
            name[0] = '\0';
            token[0] = '\0';
            parenopen = 0;
            parenclose = 0;
            dcl(); /* parse rest of line */
            if (parenopen > 0) {
```

```
                printf("error: missing one or more )
\n");
            }
            if (parenclose > 0) {
                printf("error: missing one or more
(\n");
            }
            printf("%s: %s %s\n", name, out,
datatype);
        }
    }
    return 0;
}

/* dcl: parse a declarator */
static void dcl(void)
{
    int ns;
    for (ns = 0; gettoken() == '*'; ) /* count *'s */
        ns++;
    dirdcl();
    while (ns-- > 0)
        strcat(out, " pointer to");
}

/* dirdcl: parse a direct declarator */
static void dirdcl(void)
{
    int tempparens;
    if (tokentype == ','){ /* allow multiple
declarators on a line */
        tokentype = '\n';
    }
    if (tokentype == '\n'){
        return;
    }
    if (tokentype == '('){ /* ( dcl ) */
        ++parenopen;
        dcl();
        if (tokentype == ')'){
            --parenopen;
```

```
        } else {
            if(oldtoken == '(')
                strcat(out, " function returning");
            ungetch();
        }
    } else if (tokentype == NAME){ /* variable name
*/
        strcpy(name, token);
    } else if (tokentype == PARENS){
        strcat(out, " function returning");
    } else if (tokentype == BRACKETS){
        strcat(out, " array");
        strcat(out, token);
        strcat(out, " of");
    } else if (oldtoken == NAME && tokentype == ')'){
        ++parenclose;
        strcat(out, " function returning");
    } else {
        printf("error: expected name or (dcl)\n");
    }
    gettoken();
    while (tokentype == PARENS || tokentype ==
BRACKETS
            || tokentype == '(' ||
isdigit(tokentype) ||
            tokentype == NAME){
        if (tokentype == PARENS){
            strcat(out, " function returning");
        } else if (tokentype == BRACKETS) {
            strcat(out, " array");
            strcat(out, token);
            strcat(out, " of");
        } else if (tokentype == '('){
            /* process function with parameters...
for now
             * just ignoring them */
            /* prevents detection of unmatched ) in
gettoken() */
            ++parenopen; /* don't remove this! */
            tempparens = 1; /* track balanced
parentheses */
```

```
            strcat(out, " function returning");
            do {
                gettoken();
                if (tokentype == '('){
                    ++tempparens;
                } else if (tokentype == ')'){
                    --tempparens;
                } else if (tokentype == '\n'){
                    return;
                }
            } while (tokentype != ')' ||
tempparens != 0);
            --parenopen;
        } else {
            /* saw NAME or digit: it could be this
was part
             * of a parameter to a function or a
subscript
             * to an array */
            do {
                gettoken();
                if (tokentype == BRACKETS){
                    strcat(out, " array");
                    strcat(out, token);
                    strcat(out, " of");
                    break; /* exit do-while */
                }
                if (tokentype == PARENS){
                    strcat(out, " function
returning");
                    break; /* exit do-while */
                }
            } while(tokentype != '\n');
            if (tokentype == '\n')
                return;
        }
        gettoken();
    }
}

/* gettoken: return next token.
```

```
 * I have getch() and ungetch() included from the
header
 * file getch.h , note that ungetch() in my
implementation
 * requires no parameter.
 */
static int gettoken(void)
{
     int c;
     int i;
     char *p = token;
     oldtoken = tokentype; /* back up tokentype */
     while ((c = getch()) == ' ' || c == '\t')
          ;
     /* remove comments */
     while (c == '/'){
          c = getch();
          if (c == '/'){
               while (c != '\n')
                    c = getch();
          }
          if (c == '*'){ /* start of comment */
               do {
                    c = getch();
                    if (c == '*'){
                         c = getch();
                         if(c == '/'){
                              c = getch();
                              break;
                         }
                    }
               } while (c != '\n');
          }
     }
     /* Assume anything past one of these characters
is something
      * we do not care to process. */
     if (c == ';' || c == '{' || c == '='){
          while (c != '\n')
               c = getch();
     }
```

```
    /* count unmatched closing parentheses */
    if (c == ')' && parenopen == 0){
        ++parenclose;
        return tokentype = PARENS;
    }
    if (c == '('){
        if ((c = getch()) == ')'){
            strcpy(token, "()");
            return tokentype = PARENS;
        } else {
            ungetch();
            return tokentype = '(';
        }
    } else if (c == '['){ /* we sort of know what
should be in here */
        alphaseen = 0;
        for (*p++ = c, i = 0; (c = getch()); ++i){
            while (c == ' ' || c == '\t') /* allow
for space */
                c = getch();
            if (i == 0 && isalpha(c)){
                *p++ = c;
                alphaseen = 1;
            } else if (isdigit(c) && alphaseen == 0)
{
                *p++ = c;
            } else if (i > 0 && isalpha(c)){
                printf("error: invalid array
subscript\n");
                while (isalnum(c = getch()))
                    ;
                ungetch();
            } else if (isdigit(c) && alphaseen){
                printf("error: invalid array
subscript\n");
                while (isalnum(c = getch()))
                    ;
                ungetch();
            } else if (c == ']'){
                *p++ = c;
                break;
```

```
            } else {
                printf("error: missing ]\n");
                *p++ = ']';
                ungetch();
                break;
            }
        }
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (c == ']'){ /* unmatched closing
bracket? */
        printf("error: missing [\n");
        *p++ = '[';
        *p++ = ']';
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (isalpha(c) || c == '_') { /* an _ is a
valid char too */
        for (*p++ = c; isalnum(c = getch()) || c ==
'_'; )
            *p++ = c;
        *p = '\0';
        ungetch();
        return tokentype = NAME;
    } else {
        return tokentype = c;
    }
}
```

getch.c file:

```
#include <stdio.h>
#define BUFSIZE 1000
static char line[BUFSIZE]; /* buffer for line */
static int bufp = 0; /* position in buf */
static int readflag = 1;
static int get_line(char *s, int max_length);
/* getch: read a character */
int getch(void)
{
    int length = 0;
    do {
        if (readflag == 1){
```

```
            /* prime the line array */
            if ((length = get_line(line,BUFSIZE)) >
0){
                if(length >= (BUFSIZE - 1)){
                    printf("ERROR: line buffer
exceeded\n");
                    return EOF;
                }
            }else{
                return EOF;
            }
            bufp = 0;
            readflag = 0;
        }
        if (line[bufp] == '\0')
            readflag = 1; /* need to read a new line
*/
    } while (readflag);
    return line[bufp++];
}
/* ungetch: push character back on input */
void ungetch(void)
{
    if (bufp > 0)
        --bufp;
}
/*
 * print_source: print out the contents of the line
read
 */
void print_source(void)
{
    printf("%s", line);
}
/*
 * getline:
 * Author: Robert Taylor
 * This version will read a line up to a maximum
number of characters
 * (max_length - 1) and store a '\0' at the end. The
number of characters
```

```
 * read is returned, if that is a 0 then we are done
reading lines.
 */
int get_line(char *s, int max_length)
{
    int c;
    char *start = s; /* save pointer to start of
buffer s */
    char *end = s + (max_length - 2); /* point near
to end of buffer s */
    while((c = getchar()) != EOF && c != '\n' && s <
end){
        *s++ = c;
    }
    /* store last character read */
    if(c != EOF)
        *s++ = c;
    *s = '\0';/* terminate the line */
    return s - start; /* number of characters read */
}


fetch.h file:
#ifndef GETCH
#define GETCH
int getch(void);
void ungetch(void);
void print_source(void);
#endif

19.

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "getch.h"
#define MAXTOKEN 100
#define MAXPOINTERS 10
/* Author: Robert Taylor
 * Date: June, 2014
 * Exercise 5-19. Modify undcl so that it does not
```

```
add redundant parentheses
 * to declarations.
 *
 * From the input syntax stipulated for undcl,
namely:
 * x () * [] * () char
 * to generate:
 * char (*(*x())[])()
 *
 * We can see that...
 * 1. The only time that parentheses are seen in the
input are if they
 * indicate a function.
 * 2. Other than when they indicate a function,
parentheses in the output are
 * only added when a pointer '*' is seen.
 *
 * Parentheses used to represent a function are never
redundant. So the item
 * of code that we need to focus on to solve the
exercise is the pointer
 * section.
 *
 * Sample: x () * * * char
 * output before changes: char(*(*(*x())))
 * output after changes: char(***x())
 */
enum { NAME, PARENS, BRACKETS };


int gettoken(void);
int tokentype; /* type of last token */
char token[MAXTOKEN]; /* last token string */
char out[1000]; /* output string */

/* undcl: convert word description to declaration */
int main(void)
{
     int type;
     int i, c;
     int pcount;
```

```
    char temp[MAXTOKEN];
    char p[MAXPOINTERS]; /* space for 9 pointers */
    while (gettoken() != EOF){
        strcpy(out, token);
        pcount = 0;
        while ((type = gettoken()) != '\n')
            if (type == PARENS || type == BRACKETS){
                strcat(out, token);
            } else if (type == '*'){
                pcount++;
                while ((c = getch()) == '*' || c ==
' '){
                    if (c == '*'){
                        if (pcount < (MAXPOINTERS -
1))
                            pcount++;
                        else
                            break;
                    }
                }
                ungetch();
                for (i = 0; i < pcount; i++){
                    p[i] = '*';
                }
                p[i] = '\0';
                pcount = 0;
                sprintf(temp, "(%s%s)", p, out);
                strcpy(out, temp);
            } else if (type == NAME){
                sprintf(temp, "%s%s", token, out);
                strcpy(out, temp);
            } else {
                printf("invalid input at %s\n",
token);
            }
        printf("%s\n", out);
    }
    return 0;
}

/* gettoken: return next token.
```

```
 * I have getch() and ungetch() included from the
header
 * file getch.h , note that ungetch() in my
implementation
 * requires no prameter.
 */
int gettoken(void)
{
    int c;
    char *p = token;
    while ((c = getch()) == ' ' || c == '\t')
        ;
    if (c == '('){
        if ((c = getch()) == ')'){
            strcpy(token, "()");
            return tokentype = PARENS;
        } else {
            ungetch();
            return tokentype = '(';
        }
    } else if (c == '['){
        for (*p++ = c; (*p++ = getch()) != ']'; )
            ;
        *p = '\0';
        return tokentype = BRACKETS;
    } else if (isalpha(c)) {
        for (*p++ = c; isalnum(c = getch()); )
            *p++ = c;
        *p = '\0';
        ungetch();
        return tokentype = NAME;
    } else {
        return tokentype = c;
    }
}
```

getch.c source

```
#include <stdio.h>
#define BUFSIZE 1000
static char line[BUFSIZE]; /* buffer for line */
static int bufp = 0; /* position in buf */
static int readflag = 1;
```

```c
static int get_line(char *s, int max_length);
/* getch: read a character */
int getch(void)
{
    int length = 0;
    do {
        if (readflag == 1){
            /* prime the line array */
            if ((length = get_line(line,BUFSIZE)) >
0){
                if(length >= (BUFSIZE - 1)){
                    printf("ERROR: line buffer
exceeded\n");
                    return EOF;
                }
            }else{
                return EOF;
            }
            bufp = 0;
            readflag = 0;
        }
        if (line[bufp] == '\0')
            readflag = 1; /* need to read a new line
*/
    } while (readflag);
    return line[bufp++];
}
/* ungetch: push character back on input */
void ungetch(void)
{
    if (bufp > 0)
        --bufp;
}
/*
 * print_source: print out the contents of the line
read
 */
void print_source(void)
{
    printf("%s", line);
}
```

```
/*
 * getline:
 * Author: Robert Taylor
 * This version will read a line up to a maximum
number of characters
 * (max_length - 1) and store a '\0' at the end. The
number of characters
 * read is returned, if that is a 0 then we are done
reading lines.
 */
int get_line(char *s, int max_length)
{
    int c;
    char *start = s; /* save pointer to start of
buffer s */
    char *end = s + (max_length - 2); /* point near
to end of buffer s */
    while((c = getchar()) != EOF && c != '\n' && s <
end){
        *s++ = c;
    }
    /* store last character read */
    if(c != EOF)
        *s++ = c;
    *s = '\0';/* terminate the line */
    return s - start; /* number of characters read */
}
```

getch.h file:

```
#ifndef GETCH
#define GETCH
int getch(void);
void ungetch(void);
void print_source(void);
#endif
```

***chapter 6.***

*1.Our version of `getword` does not properly handle underscores, string constants, comments, or preprocessor control lines. Write a better version.*

```
/* K&R 6-1: "Our version of getword() does not
properly handle
   underscores, string constants, or preprocessor
control lines.
   Write a better version."

   This is intended to be a solution to K&R 6-1 in
"category 0" as
   defined by the official rules given on Richard
Heathfield's "The C
   Programming Language Answers To Exercises" page,
found at
   http://users.powernet.co.uk/eton/kandr2/
index.html.

   For more information on the language for which
this is a lexical
   analyzer, please see the comment preceding
getword() below.

   Note that there is a small modification to
ungetch() as defined by
   K&R.  Hopefully this lies within the rules. */

/* knr61.c - answer to K&R2 exercise 6-1.
   Copyright (C) 2000 Ben Pfaff <blp@gnu.org>.

   This program is free software; you can
redistribute it and/or
   modify it under the terms of the GNU General
Public License as
   published by the Free Software Foundation; either
version 2 of the
   License, or (at your option) any later version.
```

```
   This program is distributed in the hope that it
will be useful, but
   WITHOUT ANY WARRANTY; without even the implied
warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.  See the GNU
   General Public License for more details.

   You should have received a copy of the GNU General
Public License
   along with this program; if not, write to the Free
Software
   Foundation, Inc., 59 Temple Place - Suite 330,
Boston, MA
   02111-1307, USA. */
#include <ctype.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Tokens.  Other non-whitespace characters self-
represent themselves
   as tokens. */
enum token
  {
    TOK_ID = UCHAR_MAX + 1,     /* Identifier. */
    TOK_STRING,                 /* String constant.
*/
    TOK_CHAR,                   /* Character
constant. */
    TOK_EOF                     /* End of file. */
  };

enum token getword (char *word, int lim);

static int skipws (void);
static int getstelem (char **, int *, int);

static int getch (void);
static void ungetch (int);
```

```
static void putch (char **, int *, int);

/* Main program for testing. */
int
main (void)
{
  ungetch ('\n');

  for (;;)
    {
      char word[64];
      enum token token;

      /* Get token. */
      token = getword (word, sizeof word);

      /* Print token type. */
      switch (token)
        {
        case TOK_ID:
          printf ("id");
          break;

        case TOK_STRING:
          printf ("string");
          break;

        case TOK_CHAR:
          printf ("char");
          break;

        case TOK_EOF:
          printf ("eof\n");
          return 0;

        default:
          printf ("other");
          word[0] = token;
          word[1] = '\0';
          break;
        }
```

```
      /* Print token value more or less
unambiguously. */
      {
        const char *s;

        printf ("\t'");
        for (s = word; *s != '\0'; s++)
          if (isprint (*s) && *s != '\'')
            putchar (*s);
          else if (*s == '\'')
            printf ("\\'");
          else
            /* Potentially wrong. */
            printf ("\\x%02x", *s);
        printf ("'\n");
      }
    }
}

/* Parses C-like tokens from stdin:

        - Parses C identifiers and string and
character constants.

        - Other characters, such as operators,
punctuation, and digits
          not part of identifiers are considered as
tokens in
          themselves.

        - Skip comments and preprocessor control
lines.

   Does not handle trigraphs, line continuation with
\, or numerous
   other special C features.

   Returns a token type.  This is either one of TOK_*
above, or a single
   character in the range 0...UCHAR_MAX.
```

```
   If TOK_ID, TOK_STRING, or TOK_CHAR is returned,
WORD[] is filled
   with the identifier or string value, truncated at
LIM - 1
   characters and terminated with '\0'.

   For other returned token types, WORD[] is
indeterminate. */
enum token
getword (char *word, int lim)
{
  int beg_line, c;

  for (;;)
    {
      beg_line = skipws ();
      c = getch ();

      if (!beg_line || c != '#')
        break;

      /* Skip preprocessor directive. */
      do
        {
          c = getch ();
          if (c == EOF)
            return TOK_EOF;
        }
      while (c != '\n');
      ungetch ('\n');
    }

  if (c == EOF)
    return TOK_EOF;
  else if (c == '_' || isalpha ((unsigned char) c))
    {
      do
        {
          putch (&word, &lim, c);
          c = getch ();
```

```
        }
      while (isalnum ((unsigned char) c) || c ==
'_');

      ungetch (c);
      return TOK_ID;
    }
  else if (c == '\'' || c == '"')
    {
      int quote = c;
      word[0] = '\0';
      while (getstelem (&word, &lim, quote))
        ;
      return quote == '\'' ? TOK_CHAR : TOK_STRING;
    }
  else
    return (unsigned char) c;
}

/* Skips whitespace and comments read from stdin.
   Returns nonzero if a newline was encountered,
indicating that we're
   at the beginning of a line. */
static int
skipws (void)
{
  /* Classification of an input character. */
  enum class
    {
      CLS_WS = 0,               /* Whitespace. */
      CLS_BEG_CMT,              /* Slash-star
beginning a comment. */
      CLS_END_CMT,              /* Star-slash ending
a comment. */
      CLS_OTHER,                /* None of the above.
*/

      CLS_IN_CMT = 4            /* Combined with one
of the above,
                                   indicates we're
inside a comment. */
```

```
    };

  /* Either 0, if we're not inside a comment,
     or CLS_IN_CMT, if we are inside a comment. */
  enum class in_comment = 0;

  /* Have we encountered a newline outside a comment?
*/
  int beg_line = 0;

  for (;;)
    {
      int c;                      /* Input character.
*/
      enum class class;           /* Classification of
`c'. */

      /* Get an input character and determine its
classification. */
      c = getch ();
      switch (c)
        {
        case '\n':
          if (!in_comment)
            beg_line = 1;
          /* Fall through. */

        case ' ': case '\f': case '\r': case '\t':
case '\v':
          class = CLS_WS;
          break;

        case '/':
          /* Outside a comment, slash-star begins a
comment. */
          if (!in_comment)
            {
              c = getch ();
              if (c == '*')
                class = CLS_BEG_CMT;
              else
```

```
                {
                  ungetch (c);
                  c = '/';
                  class = CLS_OTHER;
                }
            }
          else
            class = CLS_OTHER;
          break;

        case '*':
          /* Inside a comment, star-slash ends the
comment. */
          if (in_comment)
            {
              c = getch ();
              if (c == '/')
                class = CLS_END_CMT;
              else
                {
                  ungetch (c);
                  class = CLS_OTHER;
                }
            }
          else
            class = CLS_OTHER;
          break;

        default:
          /* Other characters. */
          if (c == EOF)
            return 0;
          class = CLS_OTHER;
        }

      /* Handle character `c' according to its
classification
         and whether we're inside a comment. */
      switch (class | in_comment)
        {
        case CLS_WS:
```

```
        case CLS_WS | CLS_IN_CMT:
        case CLS_OTHER | CLS_IN_CMT:
          break;

        case CLS_BEG_CMT:
          in_comment = CLS_IN_CMT;
          break;

        case CLS_OTHER:
          ungetch (c);
          return beg_line;

        case CLS_END_CMT | CLS_IN_CMT:
          in_comment = 0;
          break;

        case CLS_BEG_CMT | CLS_IN_CMT:
        case CLS_END_CMT:
        default:
          printf ("can't happen\n");
          break;
        }
    }
}

/* Get a character inside a quoted string or
character constant.
   QUOTE is ' for a character constant or " for a
quoted string.
   *WORDP points to a string being constructed that
has *LIMP bytes
   available. */
static int
getstelem (char **wordp, int *limp, int quote)
{
  int c;

  /* Handle end-of-quote and EOF. */
  c = getch ();
  if (c == quote || c == EOF)
    return 0;
```

```
  /* Handle ordinary string characters. */
  if (c != '\\')
    {
      putch (wordp, limp, c);
      return 1;
    }

  /* We're in a \ escape sequence.
     Get the second character. */
  c = getch ();
  if (c == EOF)
    return 0;

  /* Handle simple single-character escapes. */
  {
    static const char escapes[] = {"''??\"\"\\\\a\ab
\bf\fn\nr\rt\tv\v"};
    const char *cp = strchr (escapes, c);
    if (cp != NULL)
      {
        putch (wordp, limp, cp[1]);
        return 1;
      }
  }

  /* Handle hexadecimal and octal escapes.
     This also handles invalid escapes by default,
     doing nothing useful with them.
     That's okay because invalid escapes generate
undefined behavior. */
  {
    unsigned char v = 0;

    if (c == 'x' || c == 'X')
      for (;;)
        {
          static const char hexits[] =
"0123456789abcdef";
          const char *p;
```

```
          c = getch ();
          p = strchr (hexits, tolower ((unsigned
char) c));
          if (p == NULL)
            break;
          v = v * 16 + (p - hexits);
        }
    else
      {
        int i;

        for (i = 0; i < 3; i++)
          {
            v = v * 8 + (c - '0');
            c = getch ();
            if (c < '0' || c > '7')
              break;
          }
      }

    putch (wordp, limp, v);
    ungetch (c);
  }

  return 1;
}

/* Capacity of putback buffer. */
#define BUFSIZE 100

/* Putback buffer. */
char buf[BUFSIZE];

/* Number of characters in putback buffer. */
int bufp = 0;

/* Retrieves and returns a character from stdin or
from the putback
   buffer.
   Returns EOF if end of file is encountered. */
int
```

```c
getch (void)
{
  return bufp > 0 ? buf[--bufp] : getchar ();
}

/* Stuffs character C into the putback buffer.
   From the caller's perspective, fails silently if
the putback buffer
   is full. */
void
ungetch (int c)
{
  if (c == EOF)
    return;

  if (bufp >= BUFSIZE)
    printf ("ungetch: too many characters\n");
  else
    buf[bufp++] = c;
}

/* Stuffs character C into buffer *WORDP, which has
*LIMP bytes
   available.
   Advances *WORDP and reduces *LIMP as appropriate.
   Drops the character on the floor if it would
overflow the buffer.
   Ensures that *WORDP is null terminated if
possible. */
static void
putch (char **wordp, int *limp, int c)
{
  if (*limp > 1)
    {
      *(*wordp)++ = c;
      (*limp)--;
    }
  if (*limp > 0)
    **wordp = '\0';
}
```

```
/*
   Local variables:
   compile-command: "checkergcc -W -Wall -ansi -
pedantic knr61.c -o knr61"
   End:
*/
```

2. *Write a program that reads a C program and prints in alphabetical order each group of variable names that are identical in the first 6 characters but different somewhere thereafter. Don't count words within strings and comments. Make 6 a parameter that can be set from the command line.*

```
/*
   Write a program that reads a C program and prints
in alphabetical order each group
   of variable names that are identical in the first
6 characters but different somewhere
   thereafter. Don't count words within strings and
comments. Make 6 a parameter that
   can be set from the command line.
   */

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

#define BUFSIZE 1000
#define MAXLEN 100

#define GROUP_MAX 1000
#define COMP_INDEX_LIMIT_DEFAULT 1

char buf[BUFSIZE];
int bufp = 0;

char *keyword_arr[]  = {"include",
"main" ,"return","int","char","void","\0"};
// Contains the list of keyword.
int keyword_count=0;
```

```c
typedef struct var{
    char word[MAXLEN];
    int count;
    struct var *left;
    struct var *right;
}variable;

// Data structure to hold the variables.



variable *root  = NULL;

/*
 All the variable with at least cmp_index_limit characters,
 which is obtained as cmd line argument(default 6)
 will be in the same group.
*/
variable groups[GROUP_MAX];
int group_count=0;

int cmp_index_limit = COMP_INDEX_LIMIT_DEFAULT;


void copy_var(variable *s,variable *t){

    strcpy(s->word,t->word);
    s->count = t->count;
    s->left = t->left;
    s->right = t->right;

}


variable *add_to_tree(variable *root,variable *p){

    if(root == NULL){
        root = (variable *) malloc(sizeof(variable));
```

```
        copy_var(root,p);
    }
    else{
        if(strcmp(p->word,root->word)<0)
            root->left = add_to_tree(root->left,p);
        else if(strcmp(p->word,root->word)>0)
            root->right = add_to_tree(root-
>right,p);
        else
            root->count++; // Same word occurring
again
    }
    return root;
}


variable *add_to_group(variable *p){

    int i=0,inserted_flag=0;
    for(;i<group_count;i++){
        if(strncmp(groups[i].word,p-
>word,cmp_index_limit)==0){
            add_to_tree(&groups[i],p);
            inserted_flag=1;
        }
    }
    if(!inserted_flag){
        copy_var(&groups[group_count],p);
        group_count++;
    }
}

// Check if find is a keyword

int bin_search_keyword_arr(char find[]){

    int low,high;
    high = keyword_count-1;
    low =0;
    while(low<=high){
        int mid = (low+high)/2;
```

```
        //printf("%s -- %s + low: %d high %d mid %d
\n",keyword_arr[mid],find,low,high,mid);
        int comp = strcmp(find,keyword_arr[mid]);
        if(comp == 0)
            return mid;
        else if(comp<0)
            high=mid-1;
        else
            low=mid+1;
    }
    return -1;
}

int getch(FILE *fp){
    return (bufp > 0)? buf[--bufp] : fgetc(fp);
}

void ungetch(int c){
    if(bufp >= BUFSIZE)
        printf("\nUngetch: Too many characters");
    else
        buf[bufp++] = c;
}


// getword returns the length of the word.
// Word can begin with an underscore.
int getword(char *word,int lim,FILE *fp){

    int c;
    char *w = word;

    while(isspace(c = getch(fp)));
    if(c==EOF)
        return -1;

    // Word begin with alpha or _
    if(isalpha(c) || c=='_')
        *w++=c;
    //Remove <*>
    if(c=='<'){
```

```
        while(c!='>')
            c = getch(fp);
    }

    //Remove comments
    if(c=='/'){
        c = getch(fp);
        if(c=='/'){
            while(c!='\n' && c!=EOF)
                c = getch(fp); // skip till end of
line.
        }
        else if(c=='*'){
            while(1){
                c = getch(fp);
                if(c == '*'){
                    c = getch(fp);
                    if(c=='/' || c==EOF)
                        break; // break on abrupt
end of file.
                }
                if(c == EOF)
                    break; // break on abrupt end of
file.
            }
        }
    }

    //Remove string constants
    if(c=='"'){
        do{
            c = getch(fp);
        }while(c!='"' && c!=EOF);
    }

    if(!isalpha(c) && c!='_'){
        *w = '\0';
        return w-word;
    }
    for(; --lim>0; w++){
        *w = getch(fp);
```

```
        if(!isalnum(*w) && *w!='_'){
            ungetch(*w);
            break;
        }
    }
    *w = '\0';
    return w-word;
}



// For using binary search
void sort_keyword_arr(){
    int i=0;
    char *t;
    for(i=0;i<keyword_count-1;i++){
        if(strcmp(keyword_arr[i],keyword_arr[i+1])>0)
{
            t = keyword_arr[i];
            keyword_arr[i] = keyword_arr[i+1];
            keyword_arr[i+1] = t;
            i = -1;
        }
    }
}

void sort_groups_arr(){
    int i=0;
    variable t;
    for(i=0;i<group_count-1;i++){
        if(strcmp(groups[i].word,groups[i+1].word)>0)
{
            t = groups[i];
            groups[i] = groups[i+1];
            groups[i+1] = t;
            i = -1;
        }
    }
}
```

```c
variable *create_node(char *w){

    variable *a = (variable *)
malloc(sizeof(variable));
    strcpy(a->word,w);
    a->count = 1; // Found one already.
    a->left = NULL;
    a->right= NULL;
    return a;
}

void traverse_tree(variable *root){

    if(root!=NULL){
        traverse_tree(root->left);
        printf("%s - Count: %d \n",root->word,root-
>count);
        traverse_tree(root->right);
    }
}

int main(int argc,char *argv[]){

    char line[MAXLEN];
    FILE *fp = fopen("t2.txt","r"); // Input file
with C program
    if(fp!=NULL){

        if(argc>1){
            cmp_index_limit = atoi(argv[1]);
        }
        // Calculate no of keywords
        int i=0;
        while(keyword_arr[i++][0]!='\0')
            keyword_count++;
        // Sort keywords for binary search

        sort_keyword_arr();

        // Sort list
        /*for(i=0;i<keyword_count;i++)
```

```
            puts(keyword_arr[i]); */


        puts("Results: ");
        while(getword(line,MAXLEN,fp)!=-1){
            if(line[0]!='\0' &&
bin_search_keyword_arr(line)==-1){
                // line should not be null and must
not be a keyword.
                //puts(line);
                variable *node = create_node(line);
                //puts(node->word);
                add_to_group(node);
            }
        }
        fclose(fp);

        // Sort groups to alphabetical order
        sort_groups_arr();
        for(i=0;i<group_count;i++){
            printf("Group - %d \n",i+1);
            traverse_tree(&groups[i]);
            putchar('\n');
        }
    }
    return 0;
}
```

*3.Write a cross-referencer that prints a list of all words in a document, and, for each word, a list of the line numbers on which it occurs. Remove noise words like "the", "and," and so on.*

```
/* Write a cross-referencer program that prints a
list of all words in a
 * document, and, for each word, a list of the line
numbers on which it
 * occurs. Remove noise words like "the", "and," and
so on.
 */
```

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

/* no such thing as strdup, so let's write one
 *
 * supplementary question: why did I call this
function dupstr,
 * rather than strdup?
 *
 */

char *dupstr(char *s)
{
  char *p = NULL;

  if(s != NULL)
  {
    p = malloc(strlen(s) + 1);
    if(p)
    {
      strcpy(p, s);
    }
  }

  return p;
}

/* case-insensitive string comparison */
int i_strcmp(const char *s, const char *t)
{
  int diff = 0;
  char cs = 0;
  char ct = 0;

  while(diff == 0 && *s != '\0' && *t != '\0')
  {
    cs = tolower((unsigned char)*s);
```

```
    ct = tolower((unsigned char)*t);
    if(cs < ct)
    {
      diff = -1;
    }
    else if(cs > ct)
    {
      diff = 1;
    }
    ++s;
    ++t;
  }

  if(diff == 0 && *s != *t)
  {
    /* the shorter string comes lexicographically
sooner */
    if(*s == '\0')
    {
      diff = -1;
    }
    else
    {
      diff = 1;
    }
  }

  return diff;
}


struct linelist
{
  struct linelist *next;
  int line;
};

struct wordtree
{
  char *word;
  struct linelist *firstline;
```

```
  struct wordtree *left;
  struct wordtree *right;
};

void printlist(struct linelist *list)
{
  if(list != NULL)
  {
    printlist(list->next);
    printf("%6d ", list->line);
  }
}

void printtree(struct wordtree *node)
{
  if(node != NULL)
  {
    printtree(node->left);
    printf("%18s  ", node->word);
    printlist(node->firstline);
    printf("\n");
    printtree(node->right);
  }
}

struct linelist *addlink(int line)
{
  struct linelist *new = malloc(sizeof *new);
  if(new != NULL)
  {
    new->line = line;
    new->next = NULL;
  }

  return new;
}

void deletelist(struct linelist *listnode)
{
  if(listnode != NULL)
  {
```

```
    deletelist(listnode->next);
    free(listnode);
  }
}

void deleteword(struct wordtree **node)
{
  struct wordtree *temp = NULL;
  if(node != NULL)
  {
    if(*node != '\0')
    {
      if((*node)->right != NULL)
      {
        temp = *node;
        deleteword(&temp->right);
      }
      if((*node)->left != NULL)
      {
        temp = *node;
        deleteword(&temp->left);
      }
      if((*node)->word != NULL)
      {
        free((*node)->word);
      }
      if((*node)->firstline != NULL)
      {
        deletelist((*node)->firstline);
      }
      free(*node);
      *node = NULL;
    }
  }
}

struct wordtree *addword(struct wordtree **node, char
*word, int line)
{
  struct wordtree *wordloc = NULL;
  struct linelist *newline = NULL;
```

```
  struct wordtree *temp = NULL;
  int diff = 0;

  if(node != NULL && word != NULL)
  {
    if(NULL == *node)
    {
      *node = malloc(sizeof **node);
      if(NULL != *node)
      {
        (*node)->left = NULL;
        (*node)->right = NULL;
        (*node)->word = dupstr(word);
        if((*node)->word != NULL)
        {
          (*node)->firstline = addlink(line);
          if((*node)->firstline != NULL)
          {
            wordloc = *node;
          }
        }
      }
    }
    else
    {
      diff = i_strcmp((*node)->word, word);
      if(0 == diff)
      {
        /* we have seen this word before! add this
line number to
         * the front of the line number list. Adding
to the end
         * would keep them in the right order, but
would take
         * longer. By continually adding them to the
front, we
         * take less time, but we pay for it at the
end by having
         * to go to the end of the list and working
backwards.
         * Recursion makes this less painful than it
```

```
might have been.
         */
        newline = addlink(line);
        if(newline != NULL)
        {
          wordloc = *node;
          newline->next = (*node)->firstline;
          (*node)->firstline = newline;
        }
      }
      else if(0 < diff)
      {
        temp = *node;
        wordloc = addword(&temp->left, word, line);
      }
      else
      {
        temp = *node;
        wordloc = addword(&temp->right, word, line);
      }
    }
  }

  if(wordloc == NULL)
  {
    deleteword(node);
  }

  return wordloc;
}

/* We can't use strchr because it's not yet been
discussed, so we'll
 * write our own instead.
 */
char *char_in_string(char *s, int c)
{
  char *p = NULL;

  /* if there's no data, we'll stop */
  if(s != NULL)
```

```
  {
    if(c != '\0')
    {
      while(*s != '\0' && *s != c)
      {
        ++s;
      }
      if(*s == c)
      {
        p = s;
      }
    }
  }

  return p;
}

/* We can't use strtok because it hasn't been
discussed in the text
 * yet, so we'll write our own.
 * To minimise hassle at the user end, let's modify
the user's pointer
 * to s, so that we can just call this thing in a
simple loop.
 */
char *tokenise(char **s, char *delims)
{
  char *p = NULL;
  char *q = NULL;

  if(s != NULL && *s != '\0' && delims != NULL)
  {
    /* pass over leading delimiters */
    while(NULL != char_in_string(delims, **s))
    {
      ++*s;
    }
    if(**s != '\0')
    {
      q = *s + 1;
      p = *s;
```

```
      while(*q != '\0' && NULL ==
char_in_string(delims, *q))
      {
        ++q;
      }

      *s = q + (*q != '\0');
      *q = '\0';
    }
  }

  return p;
}


/* return zero if this word is not a noise word,
 * or non-zero if it is a noise word
 */

int NoiseWord(char *s)
{
  int found = 0;
  int giveup = 0;

  char *list[] =
  {
    "a",
    "an",
    "and",
    "be",
    "but",
    "by",
    "he",
    "I",
    "is",
    "it",
    "off",
    "on",
    "she",
    "so",
    "the",
```

```
    "they",
    "you"
  };
  int top = sizeof list / sizeof list[0] - 1;

  int bottom = 0;

  int guess = top / 2;

  int diff = 0;

  if(s != NULL)
  {
    while(!found && !giveup)
    {
      diff = i_strcmp(list[guess], s);
      if(0 == diff)
      {
        found = 1;
      }
      else if(0 < diff)
      {
        top = guess - 1;
      }
      else
      {
        bottom = guess + 1;
      }
      if(top < bottom)
      {
        giveup = 1;
      }
      else
      {
        guess = (top + bottom) / 2;
      }
    }
  }

  return found;
}
```

```c
/*
 * Argh! We can't use fgets()! It's not discussed
until page 164.
 * Oh well... time to roll our own again...
 */

char *GetLine(char *s, int n, FILE *fp)
{
  int c = 0;
  int done = 0;
  char *p = s;

  while(!done && --n > 0 && (c = getc(fp)) != EOF)
  {
    if((*p++ = c) == '\n')
    {
      done = 1;
    }
  }

  *p = '\0';

  if(EOF == c && p == s)
  {
    p = NULL;
  }
  else
  {
    p = s;
  }

  return p;
}

/*
 * Ideally, we'd use a clever GetLine function which
expanded its
 * buffer dynamically to cope with large lines. Since
we can't use
 * realloc, and because other solutions would require
```

```
quite hefty
 * engineering, we'll adopt a simple solution - a big
buffer.
 *
 * Note: making the buffer static will help matters
on some
 * primitive systems which don't reserve much storage
for
 * automatic variables, and shouldn't break anything
anywhere.
 *
 */

#define MAXLINE 8192

int main(void)
{
  static char buffer[MAXLINE] = {0};
  char *s = NULL;
  char *word = NULL;
  int line = 0;
  int giveup = 0;
  struct wordtree *tree = NULL;

  char *delims = " \t\n\r\a\f\v!\"%^&*()_=+{}[]\
\|/,.<>:;#~?";

  while(!giveup && GetLine(buffer, sizeof buffer,
stdin) != NULL)
  {
    ++line;
    s = buffer;
    while(!giveup && (word = tokenise(&s, delims)) !=
NULL)
    {
      if(!NoiseWord(word))
      {
        if(NULL == addword(&tree, word, line))
        {
          printf("Error adding data into memory.
Giving up.\n");
```

```
          giveup = 1;
        }
      }
    }
  }

  if(!giveup)
  {
    printf("%18s  Line Numbers\n", "Word");
    printtree(tree);
  }

  deleteword(&tree);

  return 0;
}
```

4. *Write a program that prints the distinct words in its input sorted into decreasing order of frequency of occurrence. Precede each word by its count.*

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

#include <assert.h>


typedef struct WORD
{
  char *Word;
  size_t Count;
  struct WORD *Left;
  struct WORD *Right;
} WORD;


/*
  Assumptions: input is on stdin, output to stdout.
```

```
  Plan: read the words into a tree, keeping a count
of how many we have,
        allocate an array big enough to hold
Treecount (WORD *)'s
        walk the tree to populate the array.
        qsort the array, based on size.
        printf the array
        free the array
        free the tree
        free tibet (optional)
        free international shipping!
*/

#define SUCCESS                         0
#define CANNOT_MALLOC_WORDARRAY         1
#define NO_WORDS_ON_INPUT               2
#define NO_MEMORY_FOR_WORDNODE          3
#define NO_MEMORY_FOR_WORD              4


#define NONALPHA "1234567890 \v\f\n\t\r+=-*/\
\,.;:'#~?<>|{}[]`!\"$%^&()"

int ReadInputToTree(WORD **DestTree, size_t
*Treecount, FILE *Input);
int AddToTree(WORD **DestTree, size_t *Treecount,
char *Word);
int WalkTree(WORD **DestArray, WORD *Word);
int CompareCounts(const void *vWord1, const void
*vWord2);
int OutputWords(FILE *Dest, size_t Count, WORD
**WordArray);
void FreeTree(WORD *W);
char *dupstr(char *s);


int main(void)
{
  int Status = SUCCESS;
  WORD *Words = NULL;
```

```
  size_t Treecount = 0;
  WORD **WordArray = NULL;

  /* Read the words on stdin into a tree */
  if(SUCCESS == Status)
  {
    Status = ReadInputToTree(&Words, &Treecount,
stdin);
  }

  /* Sanity check for no sensible input */
  if(SUCCESS == Status)
  {
    if(0 == Treecount)
    {
      Status = NO_WORDS_ON_INPUT;
    }
  }

  /* allocate a sufficiently large array */
  if(SUCCESS == Status)
  {
     WordArray = malloc(Treecount * sizeof
*WordArray);
     if(NULL == WordArray)
     {
       Status = CANNOT_MALLOC_WORDARRAY;
     }
  }

  /* Walk the tree into the array */
  if(SUCCESS == Status)
  {
    Status = WalkTree(WordArray, Words);
  }

  /* qsort the array */
  if(SUCCESS == Status)
  {
    qsort(WordArray, Treecount, sizeof *WordArray,
CompareCounts);
```

```
  }

  /* walk down the WordArray outputting the values */
  if(SUCCESS == Status)
  {
    Status = OutputWords(stdout, Treecount,
WordArray);
  }

  /* free the word array */
  if(NULL != WordArray)
  {
    free(WordArray);
    WordArray = NULL;
  }

  /* and free the tree memory */
  if(NULL != Words)
  {
    FreeTree(Words);
    Words = NULL;
  }

  /* Error report and we are finshed */
  if(SUCCESS != Status)
  {
    fprintf(stderr, "Program failed with code %d\n",
Status);
  }
  return (SUCCESS == Status ? EXIT_SUCCESS :
EXIT_FAILURE);
}




void FreeTree(WORD *W)
{
  if(NULL != W)
  {
    if(NULL != W->Word)
```

```
    {
      free(W->Word);
      W->Word = NULL;
    }
    if(NULL != W->Left)
    {
      FreeTree(W->Left);
      W->Left = NULL;
    }
    if(NULL != W->Right)
    {
      FreeTree(W->Right);
      W->Right = NULL;
    }
  }
}


int AddToTree(WORD **DestTree, size_t *Treecount,
char *Word)
{
  int Status = SUCCESS;
  int CompResult = 0;

  /* safety check */
  assert(NULL != DestTree);
  assert(NULL != Treecount);
  assert(NULL != Word);

  /* ok, either *DestTree is NULL or it isn't (deep
huh?) */
  if(NULL == *DestTree)  /* this is the place to add
it then */
  {
    *DestTree = malloc(sizeof **DestTree);
    if(NULL == *DestTree)
    {
      /* horrible - we're out of memory */
      Status = NO_MEMORY_FOR_WORDNODE;
    }
    else
```

```
    {
      (*DestTree)->Left = NULL;
      (*DestTree)->Right = NULL;
      (*DestTree)->Count = 1;
      (*DestTree)->Word = dupstr(Word);
      if(NULL == (*DestTree)->Word)
      {
        /* even more horrible - we've run out of
memory in the middle */
        Status = NO_MEMORY_FOR_WORD;
        free(*DestTree);
        *DestTree = NULL;
      }
      else
      {
        /* everything was successful, add one to the
tree nodes count */
        ++*Treecount;
      }
    }
  }
  else  /* we need to make a decision */
  {
    CompResult = strcmp(Word, (*DestTree)->Word);
    if(0 < CompResult)
    {
      Status = AddToTree(&(*DestTree)->Left,
Treecount, Word);
    }
    else if(0 > CompResult)
    {
      Status = AddToTree(&(*DestTree)->Left,
Treecount, Word);
    }
    else
    {
      /* add one to the count - this is the same node
*/
      ++(*DestTree)->Count;
    }
  } /* end of else we need to make a decision */
```

```
  return Status;
}


int ReadInputToTree(WORD **DestTree, size_t
*Treecount, FILE *Input)
{
  int Status = SUCCESS;
  char Buf[8192] = {0};
  char *Word = NULL;

  /* safety check */
  assert(NULL != DestTree);
  assert(NULL != Treecount);
  assert(NULL != Input);

  /* for every line */
  while(NULL != fgets(Buf, sizeof Buf, Input))
  {
    /* strtok the input to get only alpha character
words */
    Word = strtok(Buf, NONALPHA);
    while(SUCCESS == Status && NULL != Word)
    {
      /* deal with this word by adding it to the tree
*/
      Status = AddToTree(DestTree, Treecount, Word);

      /* next word */
      if(SUCCESS == Status)
      {
        Word = strtok(NULL, NONALPHA);
      }
    }
  }

  return Status;
}
```

```
int WalkTree(WORD **DestArray, WORD *Word)
{
  int Status = SUCCESS;
  static WORD **Write = NULL;

  /* safety check */
  assert(NULL != Word);

  /* store the starting point if this is the first
call */
  if(NULL != DestArray)
  {
    Write = DestArray;
  }

  /* Now add this node and it's kids */
  if(NULL != Word)
  {
    *Write = Word;
    ++Write;
    if(NULL != Word->Left)
    {
      Status = WalkTree(NULL, Word->Left);
    }
    if(NULL != Word->Right)
    {
      Status = WalkTree(NULL, Word->Right);
    }
  }

  return Status;
}


/*
   CompareCounts is called by qsort. This means that
it gets pointers to the
   data items being compared. In this case the data
items are pointers too.
```

```
*/
int CompareCounts(const void *vWord1, const void
*vWord2)
{
  int Result = 0;
  WORD * const *Word1 = vWord1;
  WORD * const *Word2 = vWord2;

  assert(NULL != vWord1);
  assert(NULL != vWord2);

  /* ensure the result is either 1, 0 or -1 */
  if((*Word1)->Count < (*Word2)->Count)
  {
    Result = 1;
  }
  else if((*Word1)->Count > (*Word2)->Count)
  {
    Result = -1;
  }
  else
  {
    Result = 0;
  }

  return Result;
}


int OutputWords(FILE *Dest, size_t Count, WORD
**WordArray)
{
  int Status = SUCCESS;
  size_t Pos = 0;

  /* safety check */
  assert(NULL != Dest);
  assert(NULL != WordArray);

  /* Print a header */
  fprintf(Dest, "Total Words : %lu\n", (unsigned
```

```c
long)Count);

  /* Print the words in descending order */
  while(SUCCESS == Status && Pos < Count)
  {
    fprintf(Dest, "%10lu %s\n", (unsigned
long)WordArray[Pos]->Count, WordArray[Pos]->Word);
    ++Pos;
  }

  return Status;
}


/*
    dupstr: duplicate a string
*/
char *dupstr(char *s)
{
  char *Result = NULL;
  size_t slen = 0;

  /* sanity check */
  assert(NULL != s);

  /* get string length */
  slen = strlen(s);

  /* allocate enough storage */
  Result = malloc(slen + 1);

  /* populate string */
  if(NULL != Result)
  {
    memcpy(Result, s, slen);
    *(Result + slen) = '\0';
  }

  return Result;
}
```

5. *Write a function* `undef` *that will remove a name and definition from the table maintained by* `lookup` *and* `install` *.*

```
int undef(char * name) {
    struct nlist * np1, * np2;

    if ((np1 = lookup(name)) == NULL)  /*  name not
found  */
        return 1;

    for ( np1 = np2 = hashtab[hash(name)]; np1 !=
NULL;
          np2 = np1, np1 = np1->next ) {
        if ( strcmp(name, np1->name) == 0 ) {  /*
name found  */

            /*  Remove node from list  */

            if ( np1 == np2 )
                hashtab[hash(name)] = np1->next;
            else
                np2->next = np1->next;

            /*  Free memory  */

            free(np1->name);
            free(np1->defn);
            free(np1);

            return 0;
        }
    }

    return 1;  /*  name not found  */
}
```

6. *Implement a simple version of the* `#define` *processor (i.e., no arguments) suitable for use with C programs, based on the routines of this section. You may also find* `getch` *and* `ungetch` *helpful.*

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

#define HASHSIZE 101
#define MAXLEN 200
#define BUFSIZE 1000
#define STATE_OUT 321
#define STATE_IN_NO_NAME 322
#define STATE_IN_WITH_NAME 323


static int state = STATE_OUT; // Initial state


char buf[BUFSIZE];
int bufp = 0;


int getch(FILE *fp){
    return (bufp > 0)? buf[--bufp] : fgetc(fp);
}

void ungetch(int c){
    if(bufp >= BUFSIZE)
        printf("\nUngetch: Too many characters");
    else
        buf[bufp++] = c;
}

// Data structure to store the definitions
typedef struct list{
    struct list *next;
    char *name;
    char *defn;
}nlist;

static nlist *hashtab[HASHSIZE]; // Pointer Table
```

```c
// Hash
unsigned hash(char *s){

    unsigned hashval;

    for(hashval=0; *s!='\0'; s++)
        hashval = *s + 31 * hashval;

    return hashval%HASHSIZE;
}


//Lookup

nlist *lookup(char *s){

    nlist *np;

    for(np = hashtab[hash(s)];np!=NULL;np=np->next)
        if(strcmp(np->name,s) == 0)
            return np;
    return NULL;
}

// Install

nlist *install(char *name,char *defn){

    nlist *np;
    unsigned hashval;
    hashval = hash(name);

    if((np = lookup(name))== NULL){

        np = (nlist *) malloc(sizeof(nlist));
        if((np->name=strdup(name))==NULL)
            return NULL;
        np->next = NULL;
        if(hashtab[hashval]==NULL){
            hashtab[hashval] = np;
```

```
        } else {
            np->next = hashtab[hashval];
            hashtab[hashval] = np;
        }
    }
    else
        free((void *) np->defn);

    if((np->defn = strdup(defn))==NULL)
        return NULL;
    return np;
}


// Print the Lookup Table
void print_all_def(){
    int i=0;
    for(;i<HASHSIZE;i++){
        if(hashtab[i]!=NULL){
            nlist *p = hashtab[i];
            while(p!=NULL){
                printf("LABEL: %s DEFN: %s\n",p-
>name,p->defn);
                p = p->next;
            }
        }
    }
}

int getword(char *word,int lim,FILE *fp){

    int c;
    char *w = word;

    while(isspace(c = getch(fp)));
    if(c==EOF){
        if(state == STATE_OUT)
            return -1;
        else{
            puts("Error: Incorrect definition\n");
            return -1;
```

```
        }
    }
    // Word should be identifier in
STATE_IN_WITH_NAME to name the definition
    if(isalpha(c) || c=='_' || (state == STATE_OUT)
|| state==STATE_IN_WITH_NAME)
        *w++=c;
    if(!isalpha(c) && c!='_' && state!=STATE_OUT &&
state!=STATE_IN_WITH_NAME){
        *w = '\0';
        return w-word;
    }
    for(; --lim>0; w++){
        *w = getch(fp);
        if(state!=STATE_IN_WITH_NAME){
            // Name of definition must be a valid
identifier
            if((!isalnum(*w) && *w!='_')){
                ungetch(*w);
                break;
            }
        }
        else
            if(isspace(*w)) // Definition can be any
character
                break;
    }
    *w = '\0';
    return w-word;
}

int main(void){
    puts("\nCheck t6.txt to understand the below
output:\n");
    char line[MAXLEN];
    FILE *fp = fopen("t6.txt","r");
    char *name,*defn,*p;
    int len;
    if(fp!=NULL){
        while((len = getword(line,MAXLEN,fp)>0)){
            switch(state){
```

```
                case STATE_OUT:
                    if(strcmp(line,"#define")==0)
                        state = STATE_IN_NO_NAME;
                    else{
                        /*
                           Check if line is present
in lookup table,
                           if yes, substitute the
definition, else print as it is.
                           */
                        nlist *np;
                        if((np=lookup(line))==NULL)
                            printf("%s ",line);
                        else
                            printf("%s ",np->defn);
                    }
                    break;
                case STATE_IN_NO_NAME:
                    // Received name for definition
                    name = (char *) malloc(len);
                    strcpy(name,line);
                    state = STATE_IN_WITH_NAME;
                    break;
                case STATE_IN_WITH_NAME:
                    // Received defn for name
                    defn = (char *) malloc(len);
                    strcpy(defn,line);
                    // Update the lookup table
                    if(install(name,defn)==NULL){
                        puts("Insert Error");
                        return -1;
                    }
                    state = STATE_OUT;
                    break;
            }
        }
        // Print the Lookup Table
        printf("\nLookup Table Now: \n");
        print_all_def();
        fclose(fp);
```

```
    }
    return 0;
}
```

chapter 7.

1. *Write a program that converts upper case to lower or lower case to upper, depending on the name it is invoked with, as found in* `argv[0]`.

```
/*

  Exercise 7-1. Write a program that converts upper
case to lower case or lower case to upper,
                depending on the name it is invoked
with, as found in argv[0].


  Assumptions: The program should read from stdin,
until EOF, converting the output to stdout
               appropriately.

               The correct outputs should be :

               Program Name                Output
               lower                       stdin with
all caps converted to lower case
               upper                       stdin with
all lowercase characters converted to uppercase
               [anything else]             helpful
message explaining how to use this



*/

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

#define SUCCESS          0
#define NO_ARGV0         1
```

```
#define BAD_NAME          2


int main(int argc, char *argv[])
{
  int ErrorStatus = SUCCESS;
  int (*convert)(int c) = NULL;
  int c = 0;

  /* check that there were any arguments */
  if(SUCCESS == ErrorStatus)
  {
    if(0 >= argc)
    {
      printf("Your environment has not provided a 
single argument for the program name.\n");
      ErrorStatus = NO_ARGV0;
    }
  }

  /* check for valid names in the argv[0] string */
  if(SUCCESS == ErrorStatus)
  {
    if(0 == strcmp(argv[0], "lower"))
    {
      convert = tolower;
    }
    else if(0 == strcmp(argv[0], "upper"))
    {
      convert = toupper;
    }
    else
    {
      printf("This program performs two functions.
\n");
      printf("If the executable is named lower then 
it converts all the input on stdin to lowercase.\n");
      printf("If the executable is named upper then 
it converts all the input on stdin to uppercase.\n");
      printf("As you have named it %s it prints this 
message.\n", argv[0]);
```

```
      ErrorStatus = BAD_NAME;
    }
  }

  /* ok so far, keep looping until EOF is encountered
*/
  if(SUCCESS == ErrorStatus)
  {
    while(EOF != (c = getchar()))
    {
      putchar((*convert)(c));
    }
  }

  /* and return what happened */
  return SUCCESS == ErrorStatus ? EXIT_SUCCESS :
EXIT_FAILURE;
}
```

2. *Write a program that will print arbitrary input in a sensible way. As a minimum, it should print non-graphic characters in octal or hexadecimal according to local custom, and break long text lines.*

```
/* Use -o for octal output, -x for hexadecimal
 */

#include <stdio.h>

#define OCTAL        8
#define HEXADECIMAL 16


void ProcessArgs(int argc, char *argv[], int *output)
{
  int i = 0;
  while(argc > 1)
  {
    --argc;
    if(argv[argc][0] == '-')
    {
```

```
      i = 1;
      while(argv[argc][i] != '\0')
      {
        if(argv[argc][i] == 'o')
        {
          *output = OCTAL;
        }
        else if(argv[argc][i] == 'x')
        {
          *output = HEXADECIMAL;
        }
        else
        {
          /* Quietly ignore unknown switches, because
we don't want to
           * interfere with the program's output.
Later on in the
           * chapter, the delights of fprintf(stderr,
"yadayadayada\n")
           * are revealed, just too late for this
exercise.
           */
        }
      ++i;
      }
    }
  }
}

int can_print(int ch)
{
  char *printable =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890 !\"#%&'()*+,-./:;<=>?[\\]^_{|}~\t\f\v\r
\n";

  char *s;
  int found = 0;

  for(s = printable; !found && *s; s++)
  {
```

```
    if(*s == ch)
    {
      found = 1;
    }
  }

  return found;
}

int main(int argc, char *argv[])
{
  int split = 80;
  int output = HEXADECIMAL;
  int ch;
  int textrun = 0;
  int binaryrun = 0;
  char *format;
  int width = 0;

  ProcessArgs(argc, argv, &output);

  if(output == HEXADECIMAL)
  {
    format = "%02X ";
    width = 4;
  }
  else
  {
    format = "%3o ";
    width = 4;
  }

  while((ch = getchar()) != EOF)
  {
    if(can_print(ch))
    {
      if(binaryrun > 0)
      {
        putchar('\n');
        binaryrun = 0;
        textrun = 0;
```

```
      }
      putchar(ch);
      ++textrun;
      if(ch == '\n')
      {
        textrun = 0;
      }

      if(textrun == split)
      {
        putchar('\n');
        textrun = 0;
      }
    }
    else
    {
      if(textrun > 0 || binaryrun + width >= split)
      {
        printf("\nBinary stream: ");
        textrun = 0;
        binaryrun = 15;
      }
      printf(format, ch);
      binaryrun += width;
    }
  }

  putchar('\n');

  return 0;
}
```

3. *Revise* `minprintf` *to handle more of the other facilities of* `printf`.

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

/* minprintf:  minimal printf with variable argument
list */
void minprintf(char *fmt, ...)
```

```
{
    va_list ap;
    char *p, *sval;
    int ival;
    double dval;
    unsigned uval;

    va_start(ap, fmt);     /* make ap point to the
first unnamed arg */
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 'd':
        case 'i':
            ival = va_arg(ap, int);
            printf("%d", ival);
            break;
        case 'c':
            ival = va_arg(ap, int);
            putchar(ival);
            break;
        case 'u':
            uval = va_arg(ap, unsigned int);
            printf("%u", uval);
            break;
        case 'o':
            uval = va_arg(ap, unsigned int);
            printf("%o", uval);
            break;
        case 'x':
            uval = va_arg(ap, unsigned int);
            printf("%x", uval);
            break;
        case 'X':
            uval = va_arg(ap, unsigned int);
            printf("%X", uval);
            break;
        case 'e':
```

```
            dval = va_arg(ap, double);
            printf("%e", dval);
            break;
        case 'f':
            dval = va_arg(ap, double);
            printf("%f", dval);
            break;
        case 'g':
            dval = va_arg(ap, double);
            printf("%g", dval);
            break;
        case 's':
            for (sval = va_arg(ap, char *); *sval;
sval++)
                putchar(*sval);
            break;
        default:
            putchar(*p);
            break;
        }
    }
    va_end(ap);
}

/* end of function */
```

4. It seems that the real scanf doesn't handle floats and strings the way I expect it to. Having said that, the book's example of a rudimentary calculator on page 141 does not work on my machine.

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

void minscanf(char *fmt, ...);

int main()
{
  int i;

  minscanf("%d", &i); /* scan integer from stdin */
  printf("scanned %d\n", i); /* print scanning
```

```
results to stdout */

  return 0;
}

/* minscanf: minimal scanf with variable argument
list
   only scans integers */
void minscanf(char *fmt, ...)
{
  va_list ap; /* points to each unnamed arg in turn
*/
  char *p;
  int *ival;

  va_start(ap, fmt); /* make ap point to 1st unnamed
arg */

  for (p = fmt; *p; p++) {

    /* skip chars that aren't format conversions */
    if (*p != '%')
      continue;

    /* prev char was %, look for format conversion */
    switch(*++p) {
    case 'd':
      ival = va_arg(ap, int *); /* get integer
pointer from args */
      scanf("%d", ival); /* read integer into int
pointer */
      break;
     default:
      break;
    }
  }
}
```

5. *Rewrite the postfix calculator of Chapter 4 to use scanf and/or sscanf to do the input and number conversion.*

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

   #define MAXOP 100 /* max size of operand or
operator */

   void push(double);
   double pop(void);

   int main()
   {
       char *c;
       char s[MAXOP], buf[MAXOP];
       double a = 0, op2;
       char e = '\0';

       while (scanf("%s%c", s, &e) == 2) { /* get no-
space string and space behind it */
             if (sscanf(s, " %lf", &a) == 1) /* is it
a number */
                 push(a);
             else if (sscanf(s, "%s", buf)) {
                 for (c = buf ; *c; c++) {
                     switch (*c) {
                     case '+':
                         push(pop() + pop());
                         break;
                     case '-':
                         op2 = pop();
                         push(pop() - op2);
                         break;
                     case '*':
                         push(pop() * pop());
                         break;
                     case '/':
                         op2 = pop();
                         if (op2 != 0.0)
                             push(pop() / op2);
                         else
                             printf("error: zero
```

```
divisor\n");
                            break;
                        default:
                            printf("Unknown command\n");
                            break;
                        }
                    } /* for */
                    if (e == '\n') /* print result */
                        printf("\t%.8g\n", pop());
                }
        }
        return 0;
    }

    #define MAXVAL 100  /* maximum depth of val stack
*/

    static int sp = 0;  /* next free stack position */
    static double val[MAXVAL]; /* value stack */

    /* push(): push f onto value stack */
    void push(double f)
    {
        if (sp < MAXVAL)
            val[sp++] = f;
        else
            printf("error: stack full, can't push %g
\n", f);
    }

    /* pop(): pop and return top value from stack */
    double pop(void)
    {
        if (sp > 0)
            return val[--sp];
        else {
            printf("error: stack empty\n");
            return 0.0;
        }
    }
```

*6.Write a program to compare two files, printing the first line where they differ.*

```
/
*****************************************************
*
   KnR 7-6
   --------
   Write a program to compare two files and print the
   first line where they differ.

   Author: Rick Dearman
   email: rick@ricken.demon.co.uk

   Note: This program prints ALL the lines that are
         different using the <> indicators used by
     the unix diff command. However this program
     will not cope with something as simple as a
     line being removed.

     In reality the program would be more useful
     if it searched forward for matching lines.
     This would be a better indicator of the simple
     removal of some lines.

     This has lead Richard Heathfield to track down a
version of the
     "diff" command available on GNU/Linux systems.
     for more information go to the web site at:
     www.gnu.org

*****************************************************
*/
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

void diff_line( char *lineone, char *linetwo, int
linenumber )
```

```
{
  if(strcmp (lineone, linetwo) < 0 || strcmp
(lineone, linetwo) > 0)
    printf( "%d<%s\n%d>%s\n", linenumber, lineone,
linenumber, linetwo);
}

int main(int argc, char *argv[] )
{
  FILE *fp1, *fp2;
  char fp1_line[MAXLINE], fp2_line[MAXLINE];
  int i;

  if ( argc != 3 )
    {
      printf("differ fileone filetwo\n");
      exit(0);
    }

  fp1 = fopen( argv[1], "r" );
  if ( ! fp1 )
    {
      printf("Error opening file %s\n", argv[1]);
    }

  fp2 = fopen( argv[2], "r" );
  if ( ! fp2 )
    {
      printf("Error opening file %s\n", argv[2]);
    }
  i = 0;
  while ( (fgets(fp1_line, MAXLINE, fp1) != NULL)
      && (fgets(fp2_line, MAXLINE, fp2) != NULL))
  {
    diff_line( fp1_line, fp2_line, i );
    i++;
  }

  return 0;
}
```

7. *Modify the pattern finding program of Chapter 5 to take its input from a set of named files or, if no files are named as arguments, from the standard input. Should the file name be printed when a matching line is found?*

```
/*
 * K&R2 exercise 7-7    By: Barrett Drawdy
 *
 * Modify the pattern finding program of Chapter 5
(on page 117) to take its
 * input from a set of named files or, if no files
are named as arguments,
 * from the standard input.  Should the file name be
printed when a matching
 * file is found?
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXFILES 10  /* maximum number of files to
search in */
#define MAXLINE 1024 /* longest line that can be read
at once + 1 for '\0' */

struct file {
    FILE *p;
    char *name;
};

int main(int argc, char *argv[])
{
    struct file files[MAXFILES + 1];
    struct file *fp = files;
    char **argp;
    char *pat;
    char line[MAXLINE];
    int c;
    int found  = 0, except = 0, number = 0;
```

```
    long line_num;

    if(argc < 2) {
        fprintf(stderr, "usage: %s -x -n [file1]
[file2] ... pattern\n",
                argv[0]);
        exit(-1);
    }

    /* get pattern */
    pat = argv[--argc];

    /* open files and read arguments */
    for(argp = argv + 1; argp - argv < argc; ++argp)
{
        /* read arguments */
        if(*argp[0] == '-') {
            while((c = *++argp[0]))
                switch(c) {
                case 'x':
                    except = 1;
                    break;
                case 'n':
                    number = 1;
                    break;
                default:
                    fprintf(stderr, "%s: illegal
option %c\n",
                        argv[0], c);
                    fprintf(stderr,
                        "usage: %s -x -n [file1]
[file2] ... pattern\n",
                        argv[0]);
                    exit(-1);
                }
        }
        /* read filenames */
        else {
            if(fp - files >= MAXFILES) {
                fprintf(stderr, "%s: can only
open %d files\n", argv[0],
```

```
                        MAXFILES);
                exit(-1);
            }
            if((fp->p = fopen(*argp, "r")) == NULL)
{
                fprintf(stderr, "%s: error
opening %s\n", argv[0], *argp);
                exit(-1);
            }
            else
                fp++->name = *argp;
        }
    }

    /* if there were no filenames, read from stdin */
    if(fp == files) {
        fp++->p = stdin;
    }
    fp->p = NULL;    /* put NULL pointer at end of
array */

    /* search for pattern in each file */
    for(fp = files; fp->p != NULL; ++fp) {
        line_num = 0;
        while(fgets(line, MAXLINE, fp->p) != NULL) {
            ++line_num;
            if((strstr(line, pat) != NULL) !=
except) {
                if(fp->p != stdin)
                    printf("%s ", fp->name);
                if(number)
                    printf("%ld", line_num);
                if(number || fp->p != stdin)
                    putchar(':');
                puts(line);
                ++found;
            }
        }
    }

    /* clean up */
```

```
    for(fp = files; fp->p != NULL; ++fp)
        fclose(fp->p);
    return found;
} /* end of main */
```

8. *Write a program to print a set of files, starting each new one on a new page, with a title and a running page count for each file.*

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

#define LINES_PER_PAGE 10
#define TRUE           1
#define FALSE          0

void print_file(char *file_name)
{
  FILE *f;
  int page_number = 1;
  int line_count;
  int c;
  int new_page = TRUE;

  assert(file_name != NULL);

  if ((f = fopen(file_name, "r")) != NULL) {
    while ((c = fgetc(f)) != EOF) {
      if (new_page) {
        /* print out the header */
        printf("[%s] page %d starts\n", file_name,
page_number);
        new_page = FALSE;
        line_count = 1;
      }
      putchar(c);
      if (c == '\n' && ++line_count > LINES_PER_PAGE)
{
        /* print out the footer */
        printf("[%s] page %d ends\n", file_name,
```

```
page_number);
        /* skip another line so we can see it on
screen */
        putchar('\n');
        new_page = TRUE;
        page_number++;
      }
    }
    if (!new_page) {
      /* file ended in the middle of a page, so we
still need to
         print a footer */
      printf("[%s] page %d ends\n", file_name,
page_number);
    }
    /* skip another line so we can see it on screen
*/
    putchar('\n');
    fclose(f);
  }
}

int main(int argc, char *argv[])
{
  int i;

  if (argc < 2) {
    fputs("no files specified\n", stderr);
    return EXIT_FAILURE;
  }
  for (i = 1; i < argc; i++) {
    print_file(argv[i]);
  }
  return EXIT_SUCCESS;
}
```

9. *Functions like `isupper` can be implemented to save space or to save time. Explore both possibilities.*
*#1.*

```
int isupper(int c)
```

```
{
    return (c >= 'A' && c <= 'Z');
}
#2.
int isupper(int c)
{
    return (strchr("ABCDEFGHIJKLMNOPQRSTUVWXYZ", c) !
= NULL);
}

chapter 8.
```

1. *Rewrite the program* `cat` *from Chapter 7 using* `read`, `write`, `open` *and* `close` *instead of their standard library equivalents. Perform experiments to determine the relative speeds of the two versions.*

```
#include <stdio.h>
#include <fcntl.h>
#define BUFSIZE 1024


int main(int argc, char *argv[])
{
  int fd1;
  void filecopy(int f, int t);

  if(argc == 1)
    filecopy(0, 1);

  else {
    while(--argc > 0)
      if(( fd1 = open(*++argv, O_RDONLY, 0)) == -1) {
     printf("unix cat: can't open %s\n", *argv);
     return 1;
      }
      else {
     filecopy(fd1, 1);
     close(fd1);
      }
  }
```

```c
  return 0;

}

void filecopy(int from, int to)
{
  int n;
  char buf[BUFSIZE];

  while((n=read(from, buf, BUFSIZE)) > 0 )
    write(to, buf, n);
}
```

2. *Rewrite `fopen` and `_fillbuf` with fields instead of explicit bit operations. Compare code size and execution speed.*

```c
/*
   Rewrite fopen and _fillbuf with fields instead of
explicit bit operations.
   To avoid confusion, my implementation of standard
definitions have a suffix "x"


   Field based approach is time consuming as we would
have to check if each field
   is set or not while finding a free slot. Using Bit
manipulation, we just have to
   compare the particular flag bit to zero.

   Bit fields can be used in struct _flags structure
to reduce the total size of a File type.
 */

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

#define EOF (-1)
#define OPEN_MAX 20 // Maximum files that can be open
```

```c
#define PERMS 0666



// Field based approach for flags
typedef struct _flags{
    int _READ;
    int _WRITE;
    int _UNBUF;
    int _EOF;
    int _ERR;
}flags;

// File Data structre.

typedef struct _iobuf {
    int cnt;
    char *ptr;
    char *base;
    struct _flags flags;
    int fd;
}FILEx;

FILEx _iob[OPEN_MAX];


int _fillbuffx(FILEx *f);


#define getcx(p) (--(p)->cnt >= 0? (unsigned char)
*(p)->ptr++:_fillbufx(p))


//Check if Slot is empty by checking if all fields
are
//empty in the flags structure


int is_empty(struct _flags flags){
    if(!flags._READ && !flags._WRITE && !flags._UNBUF
&&
```

```
            !flags._EOF && !flags._ERR)
        return 1;
    return 0;
}



FILEx *fopenx(char *name,char *mode){

    int fd;
    FILEx *fp;
    // Invalid Input
    if( *mode != 'r' && *mode != 'w' && *mode !=
'a' )
        return NULL;

    // Check for free slot
    for( fp= _iob; fp< _iob + OPEN_MAX ; fp++)
        if(is_empty(fp->flags))
            break;
    // If FULL return NULL
    if( fp>= _iob+OPEN_MAX )
        return NULL;
    // Create file on Write Mode
    if( *mode == 'w')
        fd = creat(name,PERMS);
    // Append Mode - If file not present create one
    else if( *mode == 'a') {
        if((fd = open(name,O_WRONLY,0))==-1)
            fd = creat(name,PERMS);
        lseek(fd,0L,2); // Go to end, in case of
append
    }
    else
        fd = open(name,O_RDONLY,0);
    if(fd == -1)
        return NULL;
    fp->fd = fd;
    fp->cnt=0;
    if(*mode == 'r')
        fp->flags._READ = 1;
```

```
    else
        fp->flags._WRITE = 1;
    return fp;
}


int _fillbufx(FILEx *fp){

    int bufsize;

    if(fp->flags._READ == 0)
        return EOF;
    bufsize = (fp->flags._UNBUF != 0)? 1:BUFSIZ;
    if(fp->base == NULL)
        if((fp->base = (char *) malloc
(bufsize))==NULL)
            return EOF;
    fp->ptr = fp->base;
    fp->cnt = read(fp->fd,fp->ptr,bufsize);
    if(--fp->cnt< 0){

        if(fp->cnt == -1)
            fp->flags._EOF=1;
        else
            fp->flags._ERR=1;
        fp->cnt = 0;
        return EOF;
    }
    return (unsigned char) *fp->ptr++;
}


int main(void){
    //Use your own file.
    FILEx *fp = fopenx("test.c","r");
    if(fp!=NULL){
        char c;
        // getcx is a macro defined above
        while((c=getcx(fp))!=EOF)
```

```
            putchar(c);
    }
    else
        puts("Error");
    return 0;
}
```

3. *Design and write* `_flushbuf`, `fflush`, *and* `fclose`.

```
/* Editor's note: Gregory didn't supply a main() for
this. Normally, in these situations,
 * I'd supply one Richard Heathfield, so that you can
easily run and test the code. But, in this case,
 * I wouldn't know where to start! If anyone wants to
fill the gap, please let Richard Heathfield know.
 * Thanks.
 *      RJH, 28 June 2000
 */

#include <stdio.h>
/* on p.176 */
#include "syscalls.h"
/* or stdlib.h */

/* _flushbuf - flush a buffer
 * According to the code on p. 176, _flushbuf
 * is what putc calls when the buffer is full.
 * EOF as the character causes everything to
 * be written -- I don't tack on the EOF.
 */
int _flushbuf(int c, FILE *f)
{
    int num_written, bufsize;
    unsigned char uc = c;

    if ((f->flag & (_WRITE|_EOF|_ERR)) != _WRITE)
        return EOF;
    if (f->base == NULL && ((f->flag & _UNBUF) == 0))
{
```

```
        /* no buffer yet */
        if ((f->base = malloc(BUFSIZ)) == NULL)
            /* couldn't allocate a buffer, so try
unbuffered */
            f->flag |= _UNBUF;
        else {
            f->ptr = f->base;
            f->cnt = BUFSIZ - 1;
        }
    }
    if (f->flag & _UNBUF) {
        /* unbuffered write */
        f->ptr = f->base = NULL;
        f->cnt = 0;
        if (c == EOF)
            return EOF;
        num_written = write(f->fd, &uc, 1);
        bufsize = 1;
    } else {
        /* buffered write */
        if (c != EOF)
            f->ptr++ = uc;
        bufsize = (int)(f->ptr - f->base);
        num_written = write(f->fd, fp->base,
bufsize);
        f->ptr = f->base;
        f->cnt = BUFSIZ - 1;
    }
    if (num_written == bufsize)
        return c;
    else {
        f->flag |= _ERR;
        return EOF;
    }
}

/* fflush */
int fflush(FILE *f)
{
    int retval;
    int i;
```

```
    retval = 0;
    if (f == NULL) {
        /* flush all output streams */
        for (i = 0; i < OPEN_MAX; i++) {
            if ((_iob[i]->flag & _WRITE) &&
(fflush(_iob[i]) == -1))
                retval = -1;
        }
    } else {
        if ((f->flag & _WRITE) == 0)
            return -1;
        _flushbuf(EOF, f);
        if (f->flag & _ERR)
            retval = -1;
    }
    return retval;
}

/* fclose */
int fclose(FILE *f)
{
    int fd;

    if (f == NULL)
        return -1;
    fd = f->fd;
    fflush(f);
    f->cnt = 0;
    f->ptr = NULL;
    if (f->base != NULL)
        free(f->base);
    f->base = NULL;
    f->flag = 0;
    f->fd = -1;
    return close(fd);
}
```

4.*he standard library function*

`int fseek(FILE *fp, long offset, int origin)`
*is identical to `lseek` except that `fp` is a file pointer instead of a file descriptor and the return value is an `int` status, not a position. Write `fseek` . Make sure that your `fseek` coordinates properly with the buffering done for the other functions of the library.*

```
/*EXERCISE 8-4

I thought I'd improve 8-4 too.  I'm trying my best to
get this as close
to ISO C as possible given the restrictions that I'm
under.  (A real
implementation would have fsetpos() borrow some of
the same code.)

*/

/* Gregory Pietsch -- My category 0 solution to 8-4
*/

#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2

int fseek(FILE *f, long offset, int whence)
{
    int result;

    if ((f->flag & _UNBUF) == 0 && base != NULL) {
        /* deal with buffering */
        if (f->flag & _WRITE) {
            /* writing, so flush buffer */
            if (fflush(f))
                return EOF;  /* from 8-3 */
        } else if (f->flag & _READ) {
            /* reading, so trash buffer --
             * but I have to do some housekeeping
first
             */
            if (whence == SEEK_CUR) {
```

```
                /* fix offset so that it's from the
last
                 * character the user read (not the
last
                 * character that was actually read)
                 */
                if (offset >= 0 && offset <= f->cnt)
{
                    /* easy shortcut */
                    f->cnt -= offset;
                    f->ptr += offset;
                    f->flags &= ~_EOF; /* see below
*/
                    return 0;
                } else
                    offset -= f->cnt;
            }
            f->cnt = 0;
            f->ptr = f->base;
        }
    }
    result = (lseek(f->fd, offset, whence) < 0);
    if (result == 0)
        f->flags &= ~_EOF; /* if successful, clear
EOF flag */
    return result;
}
```

5. *Modify the* `fsize` *program to print the other information contained in the inode entry.*

```
/*
   Modify the fsize program to print the other
information contained in the inode entry.
   */

#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```c
#include <stdlib.h>
#include <dirent.h>
#include <pwd.h>


#define MAX_PATH 1024

#ifndef DIRSIZ
#define DIRSIZ 14
#endif


void dirwalk( char *dir,void (*fcn)(char *)){

    char name[MAX_PATH];
    struct dirent *dp;
    DIR *dfd;

    if((dfd = opendir(dir))==NULL){
        puts("Error: Cannot open Directory");
        return;
    }
    puts(dir);
    // Get each dir entry
    while((dp=readdir(dfd)) != NULL){
        // Skip . and .. is redundant.
        if(strcmp(dp->d_name,".") == 0
            || strcmp(dp->d_name,"..") ==0 )
            continue;
        if(strlen(dir)+strlen(dp->d_name)+2 >
sizeof(name))
            puts("Error: Name too long!");
        else{
            sprintf(name,"%s/%s",dir,dp->d_name);
            // Call fsize
            (*fcn)(name);
        }
    }
    closedir(dfd);
}
```

```
void fsize(char *name){
    struct stat stbuf;

    if(stat(name,&stbuf) == -1){
        puts("Error: Cannot get file stats!");
        return;
    }

    if((stbuf.st_mode & S_IFMT) == S_IFDIR){
        dirwalk(name,fsize);
    }
    struct passwd *pwd = getpwuid(stbuf.st_uid);
    //print file name,size and owner
    printf("%81d %s Owner: %s\n",
(int)stbuf.st_size,name,pwd->pw_name);
}



int main(int argc,char *argv[]){

    if(argc==1)
        fsize(".");
    else
        while(--argc>0)
            fsize(*++argv);
    return 0;
}
```

6. *he standard library function `calloc(n,size)` returns a pointer to `n` objects of size `size`, with the storage initialized to zero. Write `calloc`, by calling `malloc` or by modifying it.*

```
/*
    Exercise 8.6. The standard library function
calloc(n, size) returns a pointer to n objects
          of size size, with the storage initialised
to zero. Write calloc, by calling
          malloc or by modifying it.
```

```c
    Author: Bryan Williams

*/

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

/*
  Decided to re-use malloc for this because :
     1) If the implementation of malloc and the memory
management layer changes, this will be ok.
     2) Code re-use is great.

*/
void *mycalloc(size_t nmemb, size_t size)
{
  void *Result = NULL;

  /* use malloc to get the memory */
  Result = malloc(nmemb * size);

  /* and clear the memory on successful allocation */
  if(NULL != Result)
  {
    memset(Result, 0x00, nmemb * size);
  }

  /* and return the result */
  return Result;
}

/* simple test driver, by RJH */

#include <stdio.h>

int main(void)
{
  int *p = NULL;
  int i = 0;
```

```c
  p = mycalloc(100, sizeof *p);
  if(NULL == p)
  {
    printf("mycalloc returned NULL.\n");
  }
  else
  {
    for(i = 0; i < 100; i++)
    {
      printf("%08X ", p[i]);
      if(i % 8 == 7)
      {
    printf("\n");
      }
    }
    printf("\n");
    free(p);
  }

  return 0;
}
```

8. *Write a routine* `bfree(p,n)` *that will free an arbitrary block p of n characters into the free list maintained by malloc and free. By using bfree, a user can add a static or external array to the free list at any time.*

```c
/* bfree(): Exercise from K&R 8-8; Ads an arbitrary
block into the free *
** list maintained by malloc() and free() as written
by K&R in chapter 8*
** Messages the block to a format acceptable to the
free() list,with the*
** remaining bits (bytes) managed by Waiting To Be
free()ed (wtbfree()) *
** wtbfree() can be an empty function with the effect
of simply         *
** discarding the chopped off bits. Only intended to
be as portable as  *
** free() itself, or maybe less so due to the use of
the ALIGN macro.  */
```

```
#define ALIGN(p) (sizeof(Align)-((unsigned)(p)
%sizeof(Align)))%sizeof(Align)
/* hopelessly unportable, as p is a pointer */

void wtbfree(void *p, unsigned n);

void bfree(void *p, unsigned n)
{

    unsigned align, s, r;

    if(n < sizeof(Header)) {          /* can't free
less than this */
        wtbfree(p, n);                /* put in
WTBfree list       */
        return;
    }
    align = ALIGN(p);
    if(align) {                       /* adjust
alignment                */
        wtbfree(p, align);            /* put beginning
in WTBfree list */
        p = (char *)p + align, n -= align;
    }
    s = n / sizeof(Header), r = n % sizeof(Header);
    if(r)                             /* put trailing end
in WTBfree list   */
        wtbfree((char *)p+n-r, r);
    if(s) {                           /* if there is
something left to free */
        if (freep == NULL) {          /* Set up free list
if it's empty     */
            base.s.ptr = freep = &base;
            base.s.size = 0;
        }
        ((Header *)p)->s.size = s;
        free((Header *)p + 1);
    }
}

struct wtbheader {
```

```
    struct wtbheader *next;
    void *p;
    unsigned n;
};

void try_to_myfree(struct wtbheader *p)
{
   char *tp; unsigned align; unsigned n;

   tp = p->p, align = ALIGN(p->p);
   if(align < p->n &&
   (p->n - align) % sizeof(Header) == 0) {
       tp += align, n = p->n - align, p->n = align;
       ((Header *)tp)->s.size = n / sizeof(Header);
       free(((Header *)tp) +1);
   }
}

static struct wtbheader *headptr;

void wtbfree(void *p, unsigned n)
{
    struct wtbheader *hp, *prevp;

    if(headptr == NULL) {                         /*
first use */
        if(! (headptr = malloc(sizeof *headptr))) /*
can't save fragment, dump it */
            return;
        headptr->p = p, headptr->n = n, headptr->next
= NULL;
    }

    else if(p < headptr->p) {               /*
Special case: less than head */
        if ((char *)p + n == headptr->p) {  /* merge
*/
            headptr->p = p, headptr->n += n;
            try_to_free(headptr);
            if(!headptr->n) {                       /
* delete empty node */
```

```
                void *tp = headptr; headptr =
headptr->next;
                free(tp);
            }
        }
        else {
            struct wtbheader *tp;
            if(! (tp = malloc(sizeof *tp)))/* can't
save fragment, dump it */
                return;
            tp->p = p, tp->n = n;
            tp->next = headptr,    headptr = tp;
        }
    }
    else {
        for(prevp = hp = headptr;
        hp->next && p > hp->next->p;
        prevp = hp, hp = hp->next)
            ;

        if((char*)hp->p + hp->n == p) {         /*
merge to current */
            hp->n += n;
            try_to_free(hp);
            if(!hp->n) {                        /*
delete empty node */
                if(hp == headptr)
                    headptr = NULL;
                prevp->next = hp->next;
                free(hp);
            }
        }
        else if(hp->next && (char *)p + n == hp-
>next->p) {/* merge to next */
            hp->next->p = p, hp->next->n += n;
            try_to_free(hp->next);
            if(!hp->next->n) {                      /
* delete empty node */
                void *tp = hp->next;
                hp->next = hp->next->next;
                free(tp);
```

```
            }
        }
        else {                                      /* insert
*/
            struct wtbheader *tp;
            if(! (tp = malloc(sizeof *tp)))/* can't
save fragment, dump */
                return;
            tp->p = p, tp->n = n;
            tp->next = hp->next, hp->next = tp;
        }
    }
}
```