```
/* jumble.c - takes as input an arbirary string of characters and tests
    all permutations against a dictionary which is sorted in ASCII order.
    All words found are output.
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
// Globals
#define F_OPN_FAIL      1
#define MEM_FAIL        2

static FILE *fp;
static char dictbuf[40];

// functions

static char *sortinput(const char *inp);
static void permutes(char* left, char* right);
static char *makelower(const char *input);
static int foundit(const char *test);
static void stripnl(char *line);
static FILE *safeopen(char *path, char *mode, char *wherewhat);
static void *safemalloc(size_t num_bytes, char *wherewhat);
static char *safestrdup(const char *todup, char *wherewhat);
static void fatalerror(int failure, char *wherewhat);

int main (int argc, char** argv) {
        char *chp1, *chp2, *chp3;
    if (argc != 2) {
            printf ("\n\tRequires one string of characters to be input\n\n");
            return 1;
    }
    // grab the first line from the dictionary
    /*@-onlytrans */
    fp = safeopen("/usr/local/etc/mydict", "r", "mydict");
    (void)fgets(dictbuf,39, fp);
    stripnl(dictbuf);
    /*@-unrecog */
    chp1 = strdup("");
    chp2 = makelower(argv[1]);
    chp3 = sortinput(chp2);
    permutes(chp1,chp3);
    free(chp3);
    free(chp2);
    free(chp1);
    (void)fclose(fp);
    return 0;
} // main()
char* sortinput(const char*inp){
    // sorts strings of short length
        size_t i,j;
        size_t l;
    char* result;
    result=safestrdup(inp, "strdup in sortinput");
    l=strlen(result);
    for (i=0;i<l;i++) {
            for(j=i+1;j<l;j++){
                    if ( result[j]<result[i] ){
                            char ch;
                            ch=result[i];
                            result[i]=result[j];
                            result[j]=ch;
                    }
            }
    }
    return result;
}// sortinput()
void permutes(char* left, char* right){
        /* the general idea is to extract the characters one at a
        time from the right and the tack it onto the left. Then recurse
        until the right is empty. At that time we have a permutation in
        left. The permutations are in lexical order because the initial
        string is in lexical order before this is invoked.
        */
        size_t i,lr, ll;
        char* lbuf, *rbuf, *ch;
        lr=strlen(right);
        ll=strlen(left);

        // stop the recursion here if done
        if (lr==0) {
                if (foundit(left))
                        printf("%s\n",left);
                return;
        }//if(lt...
        // make copies of left and right
        rbuf=safestrdup(right, "strdup in permutes");
```

```
                // this length will shrink by 1 char
        lbuf=(char*)safemalloc(sizeof(char)*(ll+2), "malloc in permutes");
                                        // need room for 1 more char
        ch=strdup(" ");
        for(i=0;i<lr;i++){
                // re-intialise the buffers
                strcpy(lbuf,left);
                strcpy(rbuf,right);
                strcpy(ch," ");
                ch[0]=rbuf[i];  // extract our char
                rbuf[i]='\0';   // rbuf now has 2 strings
                strcat(rbuf, &rbuf[i+1]);  // 1 string, 1 char shorter
                strcat(lbuf,ch); // ch tacked onto the left
                permutes(lbuf,rbuf); // recurse
        }//for(i...
        free(ch);
        free(lbuf);
        free(rbuf);
        return;
} // permutes()
char *makelower(const char *input){
        size_t i, l;
        char *result = safestrdup(input, "strdup in makelower");
        l=strlen(input);
        for (i=0; i<l; i++)
                result[i] = tolower(result[i]);
        return result;
}// makelower()
int foundit(const char *test){
        int check;
        stripnl(dictbuf);
        check=strcmp(dictbuf,test);
        if (check == 0) return 1;
        for (;;) {
                if (feof(fp)) return 0;
                if (check == 0) return 1;
                if (check > 0 ) return 0;
                (void)fgets(dictbuf, 39, fp);
                stripnl(dictbuf);
                check=strcmp(dictbuf,test);
        }// while
}//foundit()
void stripnl(char *line){
        size_t i, l;
        l = strlen(line);
        for(i=0; i < l; i++){
                if (line[i] == '\n') line[i] = '\0';
        }// for
        /*@-temptrans*/
        return;
}// stripnl()
FILE *safeopen(char *path, char *mode, /*@unused@*/char *wherewhat){
        FILE *sofp;
        sofp = fopen(path, mode);
        if (!(sofp)) fatalerror(F_OPN_FAIL, path);
        /*@-nullret -dependenttrans*/
        return sofp;
}// safeopen()
void *safemalloc(size_t num_bytes, char *wherewhat){
        void *p = malloc(num_bytes);
        if (!(p)) fatalerror(MEM_FAIL, wherewhat);
        /*@ -nullpass*/
        memset(p, 0, num_bytes);
        return p;
} // safemalloc()
char *safestrdup(const char *todup, char *wherewhat){
        char *p = strdup(todup);
        if (!(p)) fatalerror(MEM_FAIL, wherewhat);
        return p;
}// safestrdup()
void  fatalerror(int failure, char *wherewhat){
        int exitcode;
        switch(failure) {
                case MEM_FAIL:
                        exitcode = 3;
                        (void)puts("Could not get required memory");
                        break;
                case F_OPN_FAIL:
                        exitcode = 2;
                        (void)puts("Failed to open file");
                        break;
                default:
                        exitcode = -127; // should never happen
                        break;
        }
        perror(wherewhat);
        exit(exitcode);
}// fatalerror()
```