Jumble – unscramble jumbled words

Introduction

I gave a talk a few years ago at an MLUG meeting about programming in C. As an example I demonstrated a program I called 'jumble' because it set out to solve a puzzle called Jumbles which appeared then as now in a number of Australian news papers. I lost the original program possibly so I wrote the program again. The new version is cleaner, tighter and more flexible. There is now no arbitrary limit imposed on the length of the input scrambled string by using fixed length buffers, these being 'malloc'd on the heap. There are still practical limitations of course because for any problem involving combinations the processing time grows proportional to n! (factorial n) where n is the length of the set being processed.

Before I get down to the fine details there are a few preliminaries that must be taken care of:

- You do need to have a suitable compiler installed such as gcc.
- To understand the use of the system functions you also need to have the necessary man pages. I had to install manpages-posix and manpages-posix-dev for my EeePC system. (Try 'man 3 fopen').
- You will need an ordered English Language dictionary with US spelling because the puzzles originate in USA. I will discuss producing and maintaining the dictionary below.

What the program does.

For example:
/home/user> jumble byou
buoy
/home/user>

Producing and Maintaining the Dictionary

In my first version of jumble I created the dictionary 'mydict' like this:

```
cat /usr/lib/ispell/english.hash | strings -n3 | tr 'A-Z' 'a-z' \

| grep '^[a-z]' | sort -u > mydict
```

The example above has been taken from the 'Rute Book' Section 8.4. I added the option to strings '-n3', the default is 4 chars maximum. Back in those days it just worked.

Since then internationalisation has moved a long way and so have the dictionaries. After fruitlessly searching for english.hash and things like default.hash I had a look for stuff with the stem 'dict' in it on my EeePC. One item of definite interest was:
/opt/firefox/dictionaries/en-US.dic

Unfortunately this dictionary conatains a lot of words and what look to be wordstems followed by some kind of a code. Here is a sample:

Abbie/M Abbi/M Abbot/M abbot/MS Abbott/M abbr abbrev abbreviated/UA abbreviates/A

abbreviate/XDSNG abbreviating/A abbreviation/M Abbye/M Abby/M ABC/M Abdel/M

So using the command string:

```
cat /opt/firefox/dictionaries/en-US.dic | strings -n3 | tr 'A-Z' 'a-z' \

| grep '^[a-z]' | sort -u > mydict
```

we end up with all in lower case complete with the messy codes in mydict. To get rid of the

mess I wrote a quick and dirty fix up called 'dictfix.c' that takes 'mydict' and strips off the tail end codes and drops the result into 'newdict'. That's not the end of the story because now we have some really short word stems and some duplication so the fix is this:

```
cat newdict | sort | uniq > mydict && rm newdict
```

Either way will work, depending on what you have on your system. I have included the dictionary 'mydict' in the tarball 'jumble_progs_dictionary' that you can download if you need it. If you roll your own dictionary, then 'jumble_progs_only' is what you want.

Now this dictionary has Websters spelling and that is fine as far as it goes. The creators of the Jumble puzzles are using American spelling so it works for that purpose. But I have written another program in this series called 'xword' for the cross word buffs. And some might want to have OED or Maquarie spelling added into this file for that purpose. Now the logic of the dictionary processing in jumble requires that the dictionary be in exact lexical order. So you add new words to the dictionary this way:

```
wc -l mydict # note the number of lines in the dictionary

cp mydict mydict.bak # make a backup copy

echo newword1 >> mydict # NB >> NOT > because we are appending to the file!

...

echo newwordn >> mydict # append the nth word

wc -l mydict # the result should be n lines bigger than the first time.

cat mydict | sort -u > newdict && wc -l newdict
```

You should have the same number of lines as above, if you've lost 1 or 2 it means that you added a word or two that were already there. Make sure your new words are in place before you continue.

```
grep newword newdict # if required

# now if you only have a line or three in the dictionary it means you did this:

echo newword > mydict # just don't do this OK!

# you did make a backup copy didn't you!

# finish with

mv newdict mydict && rm mydict.bak
```

As it happens I had to add the word 'heathrow' to the version of mydict produced as above.

A Brief Discussion of The C Language

The place to learn the C language is without doubt K&R Programming in C, # ISBN-10: 0131103628 # ISBN-13: 978-0131103627 . Another reference is the online The C Library Reference Guide by Eric Huss. I will touch only briefly on the language itself because the sources above have already covered that ground. I will discuss the action of the functions of the actual program though.

Apart from that you will need to be aware that everything must be declared to the compiler before use. In particular functions may simply be declared before use and then defined elsewhere, or they may defined fully before the compiler sees the code that uses them. I prefer the former approach because I prefer to work top down and in general want to get to my main() function before I bother too much with anything else. That's not the only way to do it and there are many programs out there that will have you wading through 100's or 1000's of lines of functions berfore you see where the action starts. It's just personal preference really.

What Makes jumble.c Unscramble Jumbled Words?

What the program does is to take the users problem word and test every permutation of that set of characters against a dictionary of English language words. I am not using the words set and permutation in a strict mathematical sense here because our 'sets' can and often do have duplicated elements. Because the very definition of a permutation of our set of N letters is each of the letters taken one at a time and concatenated with the permutations of (N-1) letters it leads very naturally to invoking the 'permute' function recursively. Here it is:

```
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(lr...

        // 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()
```

And that is where all of the hard work is done. Pretty much everything else is just house keeping. Here I will introduce our main() and discuss it below:

```
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()
```

Now it just does not come much simpler than the above. The parameters argc and argv works as follows, argc has a count of the strings contained in argv. As a minimum argc will be 1 or greater because argv[0] always contains the name of the program being run, argv[1] has the

first pararmeter and so on. In this case we want exactly 1 parameter to jumble, ie the word to unscramble. So the first test causes jumble to abort if it is invoked as 'jumble' or 'jumble word1 word2'.

Next it opens the dictionary, the function 'safeopen' invokes the system function 'fopen' and takes care of any error conditions such as missing dictionary. See man 3 fopen.

After that we initialise our buffer (dictbuf) with the first word in the dictionary to be ready when the first permutation is presented to it. The function fgets returns the newline at the end of the dictionary word provided that it is less than 39 chars long. It is, the longest word in 'mydict' as I write is 23 characters long, 'electroencephalographic' as it happens. The function 'stripnl' strips the newline off the end of the dictionary word.

These two functions:

```
char *sortinput(const char*inp);

char *makelower(const char *input);
```

do just as their name suggests. Notably, by returning pointers to the altered values they can just simply be used as parameters in the permutes() function.

Of course jumble has to test each returned permutation against the dictionary and print it or them out if it/they exist. Because the jumbled letters are sorted into lexical order before we even start, the permutations are presented in lexical order also. Consequently the program only traverses the dictionary once when solving any given jumble. The function foundit() compares the test string aginst the dictionary buffer, returning 1 if it's the same, if less than the dictionary it reurns 0, and if greater than the dictionary it advances through the dictionary until a word greater than or equal to the test string is found. It then returns 1 if equal (found) or 0 (not found) if greater.

Because the buffers used in permutes() in general are created on the heap using malloc directly or indirectly via strdup, rather than using fixed length buffers, there is no arbitrary limit placed on the size of the jumbled word you can throw at it. Of course the factorial problem very quickly imposes its own limitation on what can be done.
As an aside I might mention a few statistics here. Invoking jumble on my EeePC with a 9 character word takes about 1.5 seconds, 10 takes 15 or so seconds, and 11 takes 2.5 min roughly. Consequently 12 characters will take half an hour, 13 about 7 hours and so on. The time for
'jumble electroencephalographic'
to return is (23!)/(11!) * 2.5 minutes. That works out to 1,231,362,699 years so don't bother.

The other functions, 'safeopen()', 'safemalloc()' and 'safestrdup()' simply invoke the library functions 'fopen()', 'malloc()' and 'strdup()' respectively so as to open the dictionary file, allocate memory for strings and duplicate strings. The function 'strdup' in turn uses malloc to create space for the string it is to duplicate. In the event of failure of any of these three a NULL pointer is returned. The 'safe...' functions test the pointer and if it's NULL call 'fatalerror()' which terminates 'jumble' using the 'exit()' function. Before exiting, another library function 'perror()' is called to provide an error message based on the system global integer 'errno'. The calling functions in initialise 'wherewhat' to provide some extra diagnostic information.

Compiling the Program

Once you have the source copied into a suitable location, just cd into that directory and compile it as follows:
gcc -Wall jumble.c -o jumble

Following that, (as root; su or sudo according to taste)
mv jumble /usr/local/bin/
and
mv mydict /usr/local/etc/

Of course if these paths don't suit you all I can say is "Use the source Luke". You have it after all.

Afterwords

The dictionary as produced by the procedure stated above is not in strict characterset order. The GNU 'sort' program, or at least the version on my system (5.97) forces the apostrophe (numeric value 39) to a higher value than any of the alphabetic characters. Consequently if you try so solve a 'Jumble' which involves any of these the program will fail. I've never seen a puzzle using this character though.

I have written a sort program 'btsort' that sorts strictly in characterset order which I will introduce later.

In the listing you might notice some wierd looking comments. They are there to stop 'splint', the Debian version of 'lint', from bitching. There are not many of them because everywhere I could, I altered 'jumble.c' to make peace with 'splint'.