
The University of Queensland
School of Information Technology and Electrical Engineering
Semester One, 2009

COMP2303 / COMP7306 – Assignment 1

Due: 11pm Monday March 30, 2009

Marks: 50

Weighting: 25% of your overall assignment mark (COMP2303)

Introduction

The goal of this assignment is to give you practice at C programming. You will be building on this ability in the remainder of the course. Your task is to write a program (called `wordsearch`) which finds words from a list in a grid of letters. The program will output a list of words it has found to a file and will print (to standard output) the letters which did not form part of any words. As well as developing your C programming skills, this assignment will test your ability to follow a specific coding style guide.

This is an **individual assignment**. You should feel free to discuss aspects of C programming and the assignment specification with your fellow students, but you shouldn't actively help (or seek help from) other students with the actual coding of your assignment solution. It is cheating to look at another student's code and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you. A likely penalty for a first offence would be a mark of 0 for the assignment. Don't risk it! If you're having trouble, seek help from a member of the teaching staff – don't be tempted to copy another student's code. You should read and understand the statements on student misconduct in the course profile and on the school website:

http://www.itee.uq.edu.au/about_ITEE/policies/student-misconduct.html

Assignment Specification

Command Line Arguments

Your program (`wordsearch`) is to accept command line arguments as follows:

```
wordsearch [-sort] [-out output-filename] grid-filename list-filename
```

where italicised arguments are replaced with appropriate filenames.

In other words, your program should accept two to five command line arguments. The square brackets [] above indicate that the argument(s) is/are optional. The optional arguments must occur before the other arguments (and can be in either order if both present). The `-sort` argument (if present) means that the output list of found words should be sorted alphabetically [advanced functionality]. The `output-filename` argument is the name of the file to which the list of found words should be written. If this argument is not present, your program should output this list to a file called "found" (without the quotes) in the current directory. The `grid-filename` argument is the name of the file which contains the grid of letters (details below). The `list-filename` argument is the name of the file which contains the list of words to search for in the grid (details below).

Program Operation

Your program should read in both the grid file and the list file. It should attempt to find in the grid, each of the words in the list file. **The case of letters should not be considered when matching, e.g. “FIRE” matches “fiRE”.** Words may run horizontally (left-to-right or right-to-left), vertically (top-to-bottom or bottom-to-top) or diagonally (any direction). Every word that is found should be output to the output file. (If the `-sort` argument is given, this list should be sorted [advanced functionality].) Letters in the grid can be used in multiple words. It is possible that the grid may contain some words multiple times. In this case, the word should only be printed once. Any unused letters should be printed out to standard output – in a single line (no spaces) with a single `\n` (newline) character at the end of the line. Unused letters should be printed in left-to-right, top-to-bottom order. **These letters can be output in any case (uppercase, lowercase or mixed case).** (If there are multiple solutions to a puzzle (i.e. a word occurs multiple times in the grid), your program should just consider one instance of the word – any instance can be selected. This may mean there are different sets of unused letters possible for the puzzle.)

Grid File Format

The grid file is a text file which contains the grid of letters which will be searched. The first line of the file contains the dimension of the grid (`n`). The grid is square, so this line contains just a single number – any positive integer is acceptable, though for basic functionality you can assume `n` is no more than 20. The file contains exactly `n+1` lines (where `n` is the dimension given on the first line). The `n` lines following the dimension line each contain `n` letters (A to Z, lower case or upper case). Each line (including the dimension line and the last line) is terminated by a single `\n` (newline) character only. There are no other whitespace characters in the file. (Note that text files created with a non-UNIX text editor will probably have line endings other than a single newline.) For example, a grid file may contain the following:

```
4
FIRE
IROA
TBEB
SITS
```

List File Format

The list of words to be searched contains one or more lines (each terminated by a single `\n` (newline) character only). Each line in the file contains a single word (with letters A to Z, lower case or upper case). The words will be no longer than 40 letters (and can be larger than the grid dimension – though of course these words will never be found). No word will appear more than once in the file. There are no whitespace characters (other than newlines), punctuation characters, digits etc in the file – only letters and newline characters. For example, the list file may contain the following:

```
FITS
fire
ROE
DONKEY
FIR
```

Output File Format

Your program should output (to a file called “found”, or the name given on the command line with the `-out` argument) a list of words that have been found in the grid. The output file will have one word per line, with each line terminated by a single `\n` (newline) only. If the `-sort` argument is given on the command line, then the list should be sorted alphabetically (case is ignored) [advanced

functionality]. If the `-sort` argument is not given, then the list can be in any order.

For example, given the grid and search-list files above, the output file should contain the following (if output is sorted):

```
FIR
FIRE
FITS
ROE
```

The words can be shown in any case (upper, lower or mixed).

Examples

For the example grid and list of letters above, the string sent to standard output will be (with a trailing newline):

```
RABBITS
```

If the grid input file is as follows:

```
3
ROT
AAX
CET
```

and the word list input file contains:

```
ROT
RAT
CAT
HORSE
```

Then the output file will contain:

```
CAT
RAT
ROT
```

and

```
AXE
```

will be printed to standard output.

Errors

The table below describes how your program should respond to error conditions. The message should be printed to standard error and then the program should exit with the status given. Messages should be exactly as shown, except for status 2 and 3, where *filename* should be replaced by the name of the file (as given on the command line).

Under normal operation, your program should not print anything to standard error and should print only the unused letters to standard output and should exit with status zero.

There should not be any combination of input which causes your program to crash or loop indefinitely.

Your program can assume that the list file is correctly formatted. Error 2 below need not be produced. (We will not test your program with invalid list files.)

Error	Stderr Message (single line, followed by \n)	Exit status
Incorrect arguments.	Usage: wordsearch [-sort] [-out output-filename] grid-filename list-filename	1
The word list file does not have any words in it.	Error: empty word list file filename	2
One of the files could not be opened for read/write	Error: could not open file <i>filename</i>	3
The grid file does not start with a positive integer on a line by itself.	Error: invalid grid size	4
Any of the input lines in the grid file are too short or too long or there are not enough lines or there are invalid characters on the lines.	Error: invalid grid lines	5

Advanced functionality

Approximately ten percent of the marks for the assignment are for functionality which is more difficult to implement. Marks for this functionality can only be awarded if sufficient basic functionality is implemented in order to be able to correctly test this advanced functionality. It is not intended that most students will implement this functionality – it is more challenging than the basic functionality.

Advanced functionality will require you to use dynamic memory allocation (`malloc()` etc). This is not necessary if you only implement the basic functionality (though you can do so if you wish).

Advanced Functionality One

Implement sorting of the output list (i.e. support the `-sort` command line argument). Sorting is to be case insensitive.

Advanced Functionality Two

Support grid sizes of any size.

Other requirements

Your program must be self contained and may **not** call other programs (i.e. use of `system()`, `execl()` etc is not permitted).

Specification Updates

It is possible that this specification contains errors or inconsistencies or missing information. It is

possible that clarifications will be issued via the course website. Any such clarifications posted 5 days (120 hours) or more before the due date will form part of the assignment specification. If you find any inconsistencies or omissions, please notify the teaching staff.

A sample program will be provided that you will be able to test your program against. Details will be provided on the course website.

Hints

1. You **may** find the following functions useful. You can find the documentation via `man`.
string functions – try `man strlen` on agave
`isupper()` – for testing whether a character is upper case
`qsort()` – for sorting
`fopen()` – for opening a file
`fgets()` – for reading a line from a file
`fprintf(stderr, ...)` – for printing a message to standard error
`rewind()` – for setting a file pointer back to the beginning of a file
`fclose()` – for closing a file
`exit()` – for exiting with a given exit status.
2. A suggested approach is to build your program up in stages and test it at each stage:
 - (a) check that the command line arguments are valid (and exit with appropriate error if not);
 - (b) check that the filenames given can be opened (and exit with appropriate error if not);
 - (c) read in the grid and store it
 - (d) read in the word list and process the words one at a time
 - (e) implement a search for left-to-right horizontal words
 - (f) implement other search directions (one by one)
 - (g) implement output list sorting.
3. Note that implementation of the advanced functionality may probably require the use of dynamically allocated memory (i.e. with `malloc()` etc.). You may not need this if you are only implementing the basic functionality.

Style

You must follow **version 1.4** of the COMP2303/COMP7306 C programming style guide found at http://www.itee.uq.edu.au/~comp2303/resources/c_resources.html.

Submission

Submission must be made electronically via <http://submit.itee.uq.edu.au> under the course code COMP2303. (This applies to COMP7306 students also.) Your program should consist of just a single `.c` file and this should be the only file you submit. Only your last submission will be considered. Your submission time for the assignment will be considered to be the time of your last submission via <http://submit.itee.uq.edu.au>.

The due date for this assignment is 11pm Monday March 30, 2009. The policy on “grace days” and exceptional circumstances (e.g. illness) is as outlined in the course profile. Submissions completed before 11pm Sunday March 29, 2009 will earn additional grace days. Note that no submissions can be made more than 120 hours past the deadline (i.e. after 11pm Saturday April 4, 2009) under any circumstances.

Code Compilation

Your submitted code will be compiled on `agave.students.itee.uq.edu.au` as follows:

1. Your submitted `.c` file will be placed in a directory which is otherwise empty.

2. The file will be compiled with the command:

```
gcc -o wordsearch -Wall -ansi filename.c
```

where *filename* is replaced as appropriate.

(It is not anticipated that any additional libraries will need to be linked with your program.)

If any errors result from the commands above (i.e. the `wordsearch` executable can not be created) then you will receive 0 marks for functionality (see below). Any code without academic merit will be removed from your program before compilation is attempted (and if compilation fails, you will receive 0 marks for functionality).

Marks

Marks will be awarded for both functionality and style.

Functionality (42 marks)

Provided your code compiles (see above), you will earn functionality marks based on the number of features your program correctly implements, as outlined below. Partial marks will be awarded for partially meeting the functionality requirements. Not all features are of equal difficulty. **If your program does not allow a feature to be tested then you will receive 0 marks for that feature**, even if you claim to have implemented it. For example, if your program can never open a file then it will not be possible to determine whether your program correctly implements word matching. The markers will make no alterations to your code (other than to remove code without academic merit), although minor corrections may be possible (see penalties below). Marking will be automated – it is important that you follow the specification precisely, e.g., correctly spelling all error messages.

Program correctly handles invalid command lines	(6 marks)
Program is able to create an output file with the correct name (“found” or that given on the command line). (Contents of the file are not relevant for this criteria.)	(2 marks)
Program correctly handles being unable to open the grid file	(2 marks)
Program correctly handles being unable to open the word list file	(2 marks)
Program correctly handles being unable to open the output file	(2 marks)
Program correctly handles a grid file whose first line is not formatted correctly	(2 marks)
Program correctly handles a grid file which is incorrectly formatted (other than the first line)	(2 marks)
Program can find words running horizontally left-to-right and output remaining letters	(4 marks)
Program can find words running horizontally right-to-left and output remaining letters	(4 marks)
Program can find words running vertically (either direction) and output remaining letters	(4 marks)
Program can find words running diagonally (any direction) and output remaining letters	(6 marks)
Advanced Functionality One – program sorts words correctly when <code>-sort</code> argument is given. (Program must at least be able to correctly find all horizontal left to right words.)	(3 marks)
Advanced Functionality Two – program supports grids of any size. (Program must at least be able to correctly find all horizontal left to right words.)	(3 marks)

Style (8 marks)

Your style mark will be the minimum of:

$8 \times 0.9^{\text{(Number of style guide violations + number of compilation warnings)}}$

and

your functionality mark.

The number of compilation warnings will be the total number of distinct warning lines reported

during the compilation process described above. The number of style guide violations refers to the number of violations of version 1.4 of the COMP2303/COMP7306 C Programming Style Guide. A maximum of 5 violations will be penalised for each broad guideline area. The broad guideline areas are Naming, Comments, Braces, Whitespace, Indentation, Line Length and Overall. For naming violations, the penalty will be one violation per offending name (not per use of the name) up to the maximum of five. You should pay particular attention to commenting so that others can understand your code. The marker's decision with respect to commenting violations is final – it is the marker who has to understand your code. To satisfy layout related guidelines, you may wish to consider the `indent(1)` tool. Your style mark can never be more than your functionality mark – this prevents the submission of well styled programs which don't meet at least a minimum level of required functionality.

Penalties

In cases where submitted code does not compile due to a minor error (e.g. failure to specify a required include file), the course coordinator may make minor source code corrections and mark the “corrected” code. If this occurs, a penalty between -5 and -15 marks will be applied (at the discretion of the course coordinator) depending on the amount of code correction required. If changes are only required to one line of code, the penalty will be -5. It is recommended that you do NOT make last minute changes to your code before submission. Make sure you test your program before submission.

Late Penalties

Late penalties will apply as outlined in the course profile if no grace days remain at the time of submission.