Simple Shell

General information

This exercise will be carried in teams of five. **You have until Monday 30 January at 17:00 to email me either the group you are going to work in, or that you cannot find a group.** Anyone who fails to email by the deadline will be considered as not taking part in this exercise, i.e. we will not mark any submission from him/her. **Groups must consist of students allocated to the same lab**.

In order to get marked for the exercise, you will need to demonstrate your code working in your allocated lab. **Note that any submissions that are not demonstrated will not be marked.**

The exercise will be completed in week 11 of the semester, i.e. you have to submit all your code in myPlace and demonstrate your code in your allocated lab time by week 11. However, there is an intermediate progress check on week 7.

The exercise is organised into a number of stages. Each stage adds functionality to your shell and carries a portion of the marks. **You have to implement the stages in the order listed, i.e. you cannot get any marks for one stage if you have not done all the previous stages.** In addition to that and in order to avoid rushed/messy jobs at the end, s**tarting from week 5 for every week that passes without demonstrating at least a stage completed in the lab you will be losing the opportunity to get marks for one of the final stages**, i.e. first week of no progress you can only get marks for the first 7 stages, second week of no progress you can only get marks for the first 6 stages and so on. Moreover, **in your final submission, you cannot get marks for more than one extra stage on top of the stages completed by the lab in week 10**, even if more stages are available to you.

The marks allocated to each stage are indicative of the amount of work required for it. **This means that you should be aiming to complete stages worth about 10 to 15 marks per week. By week 7 you need to demonstrate in the lab the first two stages of the shell fully implemented. Failure to do so is an indication that you are not taking this exercise seriously and your group will be excluded from the rest of the exercise.**

You can build up your overall mark for the exercise by demonstrating to the class lecturer each stage when completed. If there are any problems with the completed stage, then you have an opportunity to improve on it and get its full marks. Note that if you complete all stages before the deadline, then you are done with the coursework for this class and you can focus on the rest of your classes for the remainder of the semester.

The exercise is marked out of 100.

There is a penalty of 10 marks for submissions where the quality of the code is poor. The code of a submission of the completed shell is considered of good quality, if all working parts are integrated into a single shell program and the code is well presented (i.e. no commented out things left behind, no test printing left, proper code formatting, etc), well organized (i.e. broken up into reasonable, cohesive functions with

no unnecessary fragmentation into files, minimal use of global variables, and reasonable variable and function names) and well documented (i.e. description in code comments of all methods, assumptions that must hold at key points of the implementation, etc).

In the demonstration of the completed shell, each group member should be able to point out what exactly their contribution to the group submission was. **Anyone absent or unable to do so, will not receive any marks for the exercise.**

There is a self and peer assessment component to the exercise that will determine the individual marks for the exercise based on the mark of the group submission. Details on how the self and peer assessment will be carried out will follow. At this stage just **note that any members that do not contribute equally to the exercise will receive a lower individual mark**.

Exercise description

The aim of the exercise is to develop a simple OS shell in C for a Unix-type system. So, you must work in a Linux environment for this exercise.

Your shell should support the following functionality:

1. Execution of external system programs including their parameters.
2. The following built-in commands:
    a. cd – change working directory
    b. getpath – print system path
    c. setpath – set system path
    d. history – print history contents (numbered list of commands in history including their parameters in ascending order from least to most recent)
    e. !! – invoke the last command from history (e.g. if your last command was ‘ls –lF’ then ‘!!’ should execute ‘ls –lF’)
    f. !<no> - invoke command with number <no> from history (e.g. !5 will execute the command with number 5 from the history)
    g. !-<no> - invoke the command with number the number of the current command minus <no> (e.g. !-3 if the current command number is 5 will execute the command with number 2, or !-1 will execute again the last command)
    h. alias – print all set aliases (alias plus aliased command)
    i. alias <name> <command> - alias name to be the command. Note that the command may also include any number of parameters, while (any number of) command parameters should work correctly with aliasing (e.g. if I alias la to be ls –la then when I type la . the shell should execute ls –la .). Note also that aliasing should also work correctly with history (e.g. !5 will execute the command with number 5 from the history if this command is an alias like the la above then !5 will execute ls –la). In the enhanced form of the alias, it should be possible to alias history invocations (e.g. if I alias ‘five’ to be !5 then when I type ‘five’ the shell should execute the command with number 5 from the history). It should also be possible to alias aliases (e.g. If I alias l to be ls and then I alias la to be l –a, then when I type la the shell should execute ls –a).

j. unalias <command> - remove any associated alias

3. Persistent history of user commands (save history in a file and load it when you run the shell again)
4. Persistent aliases (save aliases in a file and load it when you run the shell again)

Shell Execution Outline

The operation of the shell should be as follows:

Find the user home directory from the environment
Set current working directory to user home directory
Save the current path
Load history
Load aliases
Do while shell has not terminated
    Display prompt
    Read and parse user input
    While the command is a history invocation or alias then replace it with the appropriate command from history or the aliased command respectively
    If command is built-in invoke appropriate function
    Else execute command as an external process
End while
Save history
Save aliases
Restore original path
Exit

System manuals

In Unix-type systems like Linux, there are extensive manuals for all system commands, system libraries, system calls, etc. These are accessible through the 'man' command. You can see what manuals are included by typing 'man man'.

<u>Stage 1: Prompt user, and Read and parse user input, Exit shell and Initialise the working directory</u> – 10 marks

Display prompt

In most shells, the prompt can be configured by the user. Sometimes it is just a character, e.g. '>' or '$', sometimes something more sophisticated, e.g. 'current working directory>', or include the username, etc.

For your shell just keep it simple to a few characters.

The prompt is printed on the screen and the user is expected to type in the next command next to it.

Read and parse user input

The user can type pretty much anything next to the prompt. However, the input is finished when the user types <enter>. You can use the gets() function to read a whole line of text from the standard input. By making the simplifying assumption that each command is at most 512 characters long, then you know that what you will get back won't be more than that many characters long. Note that you can also use the fgets() function, which is the file reading version of gets(). Note that file versions of input/output functions can be used even when you use the standard input, output and error, by using the stdin, stdout and stderr file pointers respectively.

Parsing the input line involves breaking it into parts (tokens) based on white space (i.e. ' ' space or '\t' tab), '|', '>', '<', '&', and ';' (these symbols have a special meaning in Unix-like shells as they are used to provide output of one program as input to another, or to direct the output to a particular file, or direct the input to a particular file, or provide background execution of commands, or separate multiple commands respectively). In order to do so, you could use the strtok() function. The function uses a set of delimiters to tokenize a string (i.e. in your case to take the whole user input line and break into tokens). The function is a bit strange as it remembers previous invocations. So, by using NULL as the string you are basically saying to it to get the next token from the last used string, particularly useful for iterating through the string to retrieve all its tokens.

Continuous reading and processing of user input and exiting the shell

Your shell should continuously be prompting the user for input and reading and processing any user input until the user types 'exit' (strcmp() function can be handy here) or '<ctrl>-D' (**both are required**). Note that in the latter case, we are in fact closing the standard input, which will give an error when trying to read from it. So, you need to use this as a way of exiting the loop.

Stage 1: Testing

To test that your shell works correctly you should be able to prompt the user to enter a line of input and then print the list of tokens one at a time. It is probably a good idea to put each printed token in a single line and enclose it within a particular pair of characters (e.g. quotations) in order to be certain about the characters included in the token.

Remember to test your code extensively, i.e. with as many words as possible. Don't forget to also test what happens when you exceed the character limit or when the line is empty (i.e. just enter typed in, by the way in this case there are no tokens). In general, you should ensure that your program works correctly no matter what the user types in. This is very important for a shell as it should never crash.

You should also check that your shell keeps on working correctly no matter how many lines the user inputs.

You should finally check that your shell exits when the user types 'exit' or '<ctrl>-D'. Be careful that no segmentation faults happen on exit.

Stage 2: Execute external commands – 10 marks

In Unix-type systems, the fork() function can be used to create a new process (a child process) that is an exact duplicate of the current one. Both processes parent (original) and child (new) will continue execution from the point immediately after the fork function call.

In order to allow the child process to execute something different than the code of the parent process, we need to use one of the exec() system call variants. One of the parameters of the exec variants is the program to be executed, while another includes the parameters with which the program will be invoked.

In the case of your shell, after fork() the parent process waits (wait() function) for the child process to complete, and then continues its execution.

See Figure 3.10 in the textbook or the lecture slides for an example.

In Unix-type systems like Linux, when a user logs in or starts a terminal a shell is loaded. As part of the initialization of the shell, a number of system properties are set. These properties make up the system environment. You can see what the current environment is by typing 'setenv'.

For this stage we are only interested in one thing from the environment the PATH – the set of directories (colon separated), where the shell searches for external executable programs.

The key point for this stage is to choose an appropriate variant of exec() that will take into account the PATH environment parameter. Read also the manual very carefully about how to handle external program parameters. Remember that we should be able to support any number of command parameters.

Do not also forget to handle the situation where an invalid program was entered, in which case you should provide the user with an appropriate error message.

Parsing simplification

In order to simplify the parsing so that built-in commands can be easily added, it is a good idea to generate a single string array that includes all the tokens of the user input line. This way you can parse the user input line, in the same way, all the time irrespective of whether it contains built-in or external commands or the number of tokens it contains!

To make your life even easier you can use a fixed size array, thus putting a limit on the number of tokens accepted. Fifty tokens will be a reasonable number here.

Stage 2: Testing

First, test that the code you have added to your shell has not broken the previously working code (i.e. repeat the same tests that you carried out for stage 1). They should still work. In software engineering, this is referred to as regression testing.

To test that the new functionality in your shell works you should invoke a number of programs (system or otherwise) and see them execute.

In order to check that you are handling correctly any number of parameters a very good system program to use is 'ls' which provides a list of the contents of a directory. The good thing about it is that it takes a number of flags that change the formatting of the output and it can also take a list of directory/files (of any length) and list their contents or them.

You can check whether your program correctly handles paths (i.e. only looks for executables in the directories specified by the path) by invoking your shell from within your shell! Typically the current directory (i.e. '.') is in the path so you should be able to do this if you are running your shell from the directory the executable resides in, but not from another directory (e.g. MyShell/shell and then type 'shell' at the prompt).

Do not forget to check what happens when an invalid program is provided by the user.

Stage 3: Setting the current directory to Home, and getting and setting the path – 10 marks

For this stage, we are only interested in two things from the environment the HOME – the user's home directory, and the PATH – the list of directories within which the shell looks for executable programs.

Accessing the program environment

From a C program, we can access the environment using the getenv() function and we can change the environment using the setenv() function. If you look at the manual pages for getenv and setenv, you will find how they work (i.e. parameters needed and return values) as well as what you need to include to use them.

Keep original path

We would like to restore the path to its original value on exiting the shell as it is a good idea to put things back the way they were on exit.

We can use getenv() to get the value of the PATH. A single string will be enough to keep the original path.

Saving the path should be the first thing your shell does when it starts up.

Set the current directory to HOME

We can use getenv() to find out what the user's HOME is in order to change the current directory to it. You can change the current directory of an executing C program by using the chdir() function. Again check the manual pages for details on how to call it and what to include. Use the function to change at the start of your shell its current directory to the HOME directory.

In order to ensure that this has been done correctly, you can use the getcwd() function which gets the current working directory (check the man pages for how to call it) to print it after it has been changed.

Print and change the path – built-in commands

getpath – print system path & setpath – set system path

These two commands are about the environment PATH. The first just gets and prints its value, while the second takes a path (string of a colon-separated list of directories) as a parameter and makes it the value of PATH (note the parameters of a command just follow the command name itself). You can use getenv() and setenv() respectively for this purpose.

Your shell should also check appropriate parameters are provided by the user and print informative error messages when this is not the case. Note that there are no constraints on the format of the path placed by the environment (i.e. the system will use as a path any kind of string).

For each built-in command, it is a good idea that you have a separate function that implements it. Remember that each command can be identified by looking at the first token generated by the parsing of the command line (strcmp() function can be handy here). Each of these functions will be responsible for error checking and printing informative messages relating to the command.

If you are generating a single string array containing all the tokens of the user's input line (see parsing simplification above), then you can always call these built-in command functions with the same single parameter, i.e. the array.

Restore path

You just change the PATH environment parameter to its original value (i.e. the one you saved at the start of the shell).

Restoring the path should the last thing your shell does before it exits (remember to cover all possible ways of exiting the shell). Note that in order to confirm that this has been done correctly you must also print the path after you restored it and before you exit.

Stage 3: Testing

First, make sure that all the tests you carried out for stage 2 still work. Be careful, though, as we are now changing the directory where the shell executes and the path this will affect the execution of external programs.

What happens if you type in 'ls' now?
What happens if you type in 'myShell' now?

These are already tests for the setting of the working directory. Another test here is to print what the HOME is when you retrieve it and then print the current working directory after you changed to the HOME. In both cases, you should get the same value.

To further check your shell you could try changing the environment HOME (look at the 'set' command) and see whether your shell behaviour reflects correctly the changes you make.

To check that saving and restoring the path work, a good idea is to print the path when you save it at the beginning of the execution of the shell and then again when you exit at the end. In both cases, the printed path should be exactly the same! Be careful to ensure that you really retrieve the PATH at the end not just printing the saved value.

Following that, you should check that when getpath is called you print the current path, which should be the same as the original one.

Then you should focus on testing setpath. First, setpath the path to a new value and test that getpath prints it, then try also to see how changing the path really affects the execution of external commands (e.g. set the path to only '.' and try 'ls' or try the shell itself, etc).

Do not forget to also check that the program behaves correctly when incorrect parameters are provided to getpath and setpath.

Stage 4: Changing directories – 10 marks

cd – change working directory

This command has two forms. In the first, it has no parameters and changes the working directory to the user's home directory. In the second, it has one parameter which is a directory (either in full path or relative path form) and changes the working directory to the directory provided as a parameter. If the directory provided does not exist, then it should say that there is no such directory. In fact, you must go further and utilise the error value of the system calls using the perror() function. The shell should check appropriate parameters are used by the user and provide informative messages if this is not the case. Note, though, that any string could be considered a path as far as the command is concerned.

Remember that directory changes are effected with the chdir() function.

Remember that '.' and '..' refer to the current and the parent directory and they can be used as part of the path.

Stage 4: Testing

First, make sure that all the tests you carried out for stage 4 still work. Be careful, though, as we are now changing the current directory this will affect the execution of certain external programs.

To check the additional functionality, you can change to a directory and then call ‘pwd’ (print working directory) to see that the correct directory is printed, or call ‘ls’ to see that the contents of the correct directory are printed. You should try it out with both relative and absolute paths. You should also try simple short and complex long relative and absolute paths.

Do not forget to check that the shell behaves correctly when incorrect parameters are given to the command.

Stage 5: Adding commands to history, invoking commands from history and printing the history – 15 marks

In general, what a history feature means is that the shell remembers the instructions the user types in and it can invoke again previous commands without the user having to re-type them.

This typically works as follows:

- If the user types in ‘!<no.>’, then the shell re-executes command with number <no.>, provided that the user has already typed that many commands and they are still in history. The shell counts commands from 1 onwards and for simplicity, you could assume that the shell only remembers the last 20 commands. Note that it doesn’t matter whether the command was a built-in or external one, or even whether it was a valid command at all (i.e. if the previous command was just garbage then the shell tries to execute the garbage again). It is important that the command is executed exactly as it was typed, i.e. if the command had a number of parameters these will also have to be used when it is invoked again. For these reasons, it is a good idea to just store the full command line for each command entered to the history.
- If the user types in ‘!-<no.>’, then the shell re-executes command with the number the current command number minus <no.> provided that such a number exists (i.e. it is positive) and the command is still in history.
- If the user types in ‘!!’, then the last command from the history is executed.

Note that a command invoked from the history is not entered again into the history (i.e. only new commands are entered into the history).

Moreover, the command should be checking for appropriateness of inputs and provide the user with informative messages when the inputs are not appropriate (i.e. when not enough commands have been saved or the number is invalid).

To maintain the history inside your shell, you have two options. Either you can define a struct with an integer (the command number, starting from 1 and keep on going forever) and a string (the command line itself). As the shell only keeps the last 20 commands, you can store the history in an array of 20 struct elements. Alternatively, you can just keep an array of strings ordered from least to most recent again of size 20.

Dealing with a full array

In the case of the array of structs, one way to deal with the fact that the history array is of limited capacity is to use a circular array. In a circular array when we reach the end

of it, we just go back to the beginning. The way to do so is by taking advantage of modulo arithmetic as follows.

You keep a counter initialised to 0 that would point to the next available position in the array. When an element is entered in the array the new value of the counter is the modulo array size ('%') of the old value plus one, e.g. for an array of size 20
count = (count+1) % 20;

Note that if you keep track of the number of commands so far and the next free position in the array, this enables quite simple user input validation and error checking (i.e. you know whether the history is empty or not, and whether the history number given is valid).

An alternative way to deal with a full array is for every command that is entered in the history after the array is full, you just shift all array elements by one position, i.e. element 1 becomes 0, 2 becomes 1, etc. In the last position, you enter the most recent command.

This latter approach also works with the alternative implementation (i.e. array of strings one). In this case, you can use the array index plus one to be the index for the command to be used for printing, and analogously the number minus 1 to get the index for history invocations. Basically, in this case, commands in history are always between 1-20, while in the other cases they could be of any range of 20 number (e.g. 21-40).

Add command to history

Before we add a command in history we need to first check whether it is a history invocation or not. You can check whether this is the case, by just checking whether the first token (i.e. the command itself) starts with '!'. You may use the strcspn() function for this.

Having checked that the user input is not about history invocation, then we just add everything to the history irrespective of whether it is correct or not. The command is added to the next available place in the history array (see also *Dealing with a full array* above).

Remember that to simplify the management of the history it is a good idea to just enter the whole user input string.

Print the history – built-in command

history

The command should just go through the history data structure (i.e. the struct array mentioned above) and print for each element the number and the command line, e.g.
1 ls
2 ls –lF
3 ls …
Etc.

Invoke command from history

First, you need to check whether this is a valid history invocation or not, i.e. is the command '!!', or '!<no.>' or '!-<no.>' with a number within range. In order to do so, you need to check whether the first token is '!!' or whether it starts with '!' or '!-' (see above). In the former case, you should execute the latest command entered to history. In the latter case, for '!' the remainder of the first token (i.e. excluding the '!') will be the command number and it needs to be in range (i.e. one of the commands currently in history), while for '!-' you need the remainder of the token (i.e. exclude '!' and '-') will be the number you will subtract from the current history number and the result needs to be in range (i.e. one of the commands currently in history).

If it is a valid invocation then depending on what was typed you get from your history array the appropriate input line and you pass it back. This will be the command to be processed.

Note that your stored command will need to be parsed again for execution, but you shouldn't need to check whether it is a history invocation again, as history invocation are not part of the history itself.

If the invocation is invalid then you must print an informative error message.

Stage 5: Testing

First, make sure that all the tests you carried out for stage 4 still work.

To test the additional functionality, first, check that the history is maintained correctly. To do this type in something and then type 'history' to check that the contents of the history are correct. Keep on doing this until you fill up the history to check that the circular array or command shifting works correctly. It may be useful for facilitating testing to reduce the size of the history to much less than 20. Remember to include both simple commands and ones with multiple parameters, both built-in and external ones. Include also rubbish (remember that we don't really care what the user input is).

Then, check whether you correctly recognise history invocations and that these are not entered into the history!

Then check whether history invocations work correctly, i.e. the invoked command is re-executed. Again, include both simple commands and ones with multiple parameters, both built-in and external ones. Don't forget to include rubbish.

Finally, do not forget to check that the shell behaves correctly for incorrect history invocations (e.g. out of range ones).

Stage 6: Persistent history – 10 marks

Keeping a persistent history means that the history is saved on file and loaded between activations of the shell. For this purpose, a .hist_list file can be used. (Note: that in Unix-type systems files starting with . are considered hidden in that they are not shown in directory listings unless the user explicit asks for them).

Save history

You just write the current contents of the history data structure into the .hist_list file, overwriting its previous contents, just before the shell exits.

It is probably a good idea to follow a fairly simple format for this file, e.g. 'number command' per line.

To read or write a file in a C program you need to first open it using the fopen() function. Files are opened in a particular mode, i.e. read, write, read/write, append. Files are then processed using similar functions as those you have for processing standard input and output, typically with an f in front of the function name and a file pointer as one of the parameters. For example, scanf becomes fscanf, gets becomes fgets, printf becomes fprintf, etc.

Load history

When your shell starts it will try to open the file .hist_list (in the user's home directory), read its contents and initialise the history with them.

In your shell, you can simplify matters by considering that each command line is at most 512 characters long. In the case of the history file, this would mean that you can use the fgets function to read each line. Then you know that the first token is the command number.

Stage 6: Testing

First, make sure that all the tests you carried out for stage 5 still work.

In order to check that the saving of the history works, you can just check whether the .hist_list file is created and its contents are as expected.

In order to check that the loading of the history works, you can check whether the history contents (i.e. use the history command), are what was in the file.

Do not forget to check the behaviour of the shell under abnormal conditions (e.g. file cannot be found or fails to open, etc).

Stage 7: Setting up aliases, removing aliases, invoking aliases, and printing all aliases – 15 marks

In general, aliasing means that the shell can use a different label (usually simpler) to invoke a command. For example, users more familiar with a windows environment will set an alias for 'dir' to be 'ls –lF' in a Linux/Unix environment. In this case when the user types in 'dir' then the shell will execute 'ls –lF'. This will be the case until the user removes the alias by using the command 'unalias dir'. As was the case in the example, quite often the aliased command will also include parameters. Note also that the alias could also work with additional parameters. For example, in the case of the dir above,

the user could also type in ‘dir <a_directory>’ and will execute ‘ls –lF <a directory>’, etc.

In order to keep track of the various aliases, the shell will have to keep a data structure that maps one string (the alias) to another (the aliases command). A two-dimensional array could be used for that. Alternatively you could try to implement a more sophisticated dynamic structure, basically a map with the aliases as keys and the aliased commands as values. Unfortunately, in C, we do not have a library implementation of a map as in Java.

Note that the same command can be aliased with a number of different aliases, but of course, each alias should map to only one command. If the same alias is used again your shell should override the previous one, providing an appropriate message.

Note also that there is nothing to prevent a user from using a built-in command or an external program name as an alias. Your shell should allow this. In this case, the shell will invoke the aliased command until the alias is removed (i.e. when processing a command you first check for aliases and then for built-in commands and external programs).

Note also that at this stage it is not possible to alias an alias, only built-in commands and external programs. This basically means that you can check for an alias only once, and after that point, you treat the aliased command as a normal one (not an alias) provided by the user. At this stage, you can also alias a history invocation. This means that you don’t have to check whether it is a history invocation after you checked for an alias.

Add an alias

alias <name> <command>

The alias is the name and the aliased command is the command. Before you add the alias to the data structure that keeps them, you need to check whether the alias already exists, in which case you just override it and provide an appropriate message.

To simplify processing you can make the following assumptions. First, that the alias is a single word (contains no delimiters used in tokenization). Second, that the shell can only manage ten aliases. If you already have the maximum number of aliases set, then you should just print a message saying that no more aliases can be set. Third, you can consider the rest of the command line after the alias and any following whitespace as the aliased command.

Remove an alias

unalias <command>

If an alias command exists then it is removed. Otherwise, a message no such alias is provided. Note that if you use an array for the aliases as suggested above, then you will also need to ensure that you don’t leave any gaps in it when aliases are removed. You

could shift the elements of the array to fill any such gaps or look for gaps when trying to add a new alias.

Invoking an alias

Every command entered by the user needs to be checked on whether it is an alias or not. This should be done first before the line is fully parsed. If it is an alias you replace the alias with the aliased command and then add (concatenate) the rest of the command line to it to produce the new command line that can then be parsed. The strcat() function could be used for the concatenation.

Note that if no alias is found the original command line is parsed.

Print aliases

alias

Just prints a list of all currently set aliases. If there are no aliases set then a message is printed saying this is the case.

Stage 7: Testing

First, make sure that all the tests you carried out for stage 6 still work.

To test the added functionality you should try to set an alias for a command without parameters, for a command with parameters, for a built-in command. In each case invoke the alias and check that the invoked command is executed correctly. In the case of aliasing built-in commands, you should check that instead of the built-in command the alias is invoked. It may be helpful to have your shell print what replacement it is doing for the alias to ensure that you keep the correct contents.

Then you should try to invoke aliases with parameters like the example provided earlier and again check that it works as expected. Do not also forget to test aliasing rubbish in which case the shell should try to execute the rubbish when the alias is provided.

Second, you need to check that all aliases are kept correctly by printing them out using the alias command. At this stage, it is a good idea to delimit the alias and aliased commands, e.g. with quotes to ensure that you don't have any trailing white space or enters.

Third, you should test the removal of aliases. If an alias is removed then typing it in should be treated as any other command, i.e. if it is rubbish then the shell will attempt to execute the rubbish. An interesting thing to check is whether removal leaves any gaps in the data structure by printing all the aliases after a removal.

Finally, you should test that you handle correctly the case when the maximum number of aliases is reached.

Do not forget to check that the shell behaves correctly (i.e. informative messages are printed) when incorrect parameters are provided.

Stage 8: Persistent aliases – 10 marks

Keeping persistent aliases means that the aliases are saved on file and loaded between activations of the shell. For this purpose, a .aliases file can be used.

Save aliases

You just write the current contents of the aliases data structure into the .aliases file, overwriting its previous contents, just before the shell exits (Note: it doesn't matter whether the history or the aliases are saved first).

It is probably a good idea to follow a fairly simple format for this file, e.g.
'alias aliased command' per line.

Load aliases

When your shell starts it will try to open the file .aliases (in the user's home directory), read its contents and initialise the aliases data structure with them.

In your shell, you can simplify matters by considering that each command line is at most 512 characters long. In the case of the aliases file, this would mean that you can use the fgets function to read each line. Then you know that the first token is the alias itself and the rest of the tokens the aliased command.

Stage 8: Testing

First, make sure that all the tests you carried out for stage 7 still work.

In order to check that the saving of the aliases works, you can just check whether the .aliases file is created and its contents are as expected.

In order to check that the loading of the aliases works, you can check whether the aliases (i.e. use the alias command), are what was in the file.

Do not forget to check the behaviour of the shell under abnormal conditions (e.g. file cannot be found or fails to open, etc).

Stage 9: Enhanced aliases – 10 marks

In this stage, you should add to your implementation of the alias command the ability to alias aliases and to alias history invocations.

The main challenge here is that your input handling should keep on replacing the command token with the aliased command or command from history until you end up with a built-in command or an external program or rubbish (i.e. no more substitutions can be performed). In doing so, you should be careful about circular dependencies, i.e. scenario like alias a b, alias b a, or alias a !x, where command number x in history is a, etc. An easy way to deal with this is to set a limit on the number of substitutions carried out (e.g. 3) and give an error when that limit is exceeding. A more sophisticated way to do so would be to remember the substitutions you carried out (using something an array

or preferably something dynamic like a linked list) and before carrying out a substitution check whether the same label was encountered before in which case you give an error.

Remember that aliases and history invocation may involve parameters, not just simple commands, so these should be properly concatenated to build the final command string to be parsed for execution.

Stage 9: Testing

First, make sure that all the tests you carried out for stage 8 still work.

In order to check this works try first building chains of aliases without any cycles with commands without parameters. Then try the same but now introducing parameters at various steps in the chain.

After than build chains that mix aliases and history invocations without any cycles with command without parameters. Then try the same but not introducing parameters at various steps in the chain.

Finally check that cycles are prevented, first of aliases, without and with parameters and then also history invocations again with and without parameters.

Stage 9: Advice

As this stage is high risk, i.e. there is good chance that in trying to incorporate this functionality you may break functionality introduced in earlier stages, it is recommended that you work on it on a separate version of the shell.

It is also recommended that in order to manage the risk better you add the functionality incrementally (basically following the steps outlined for testing above). Only when you are confident that an increment works correctly then start working on the next increment keeping the increments as separate versions so that you can revert to the earlier version if things go wrong. (Note: it is a good idea to apply this incremental approach to all stages however in earlier stages it should be easier to isolate new code by commenting it out).

If you manage to get to work properly (i.e. without affecting earlier stage functionality) then you can submit it as your final version. Otherwise, you can submit two versions (one with the earlier stages working and one with your attempt at this stage).