# G52CPP Lab Exercise: Hangman Requirements (v1.0)

## 1 Overview

This is purely an exercise that you can do for your own interest. It will not affect your mark at all. You can do as little or as much of it as you wish. The ONLY reason to do this is to get feedback and to practise. However, I strongly feel that doing this will help you a LOT for the exam, as well as for the formal coursework.

This exercise involves writing a text-only turn-based puzzle game in C++. This game is a text-only version of the famous 'Hangman' game, with a slight modification. Details of hangman are available on wikipedia:
`http://en.wikipedia.org/wiki/Hangman_(game)`
You can find a graphical, playable version of a game here:
`http://theproblemsite.com/games/hangman.asp`
You can also find many other examples using the search engine of your choice.

You will create a text-only version of this game, with the ability for the user to customise it to decide how many wrong guesses they are permitted before they lose, and to change various other options using command line arguments. You will need to load the word to guess from a file, allow the player to guess the letters, mark the guess and store the result. Finally, you will also need to provide a facility to save all of the words and guesses that have been made in that session to a file.

### 1.1 Gameplay summary

In this game your program will first select a random word. It will then display the word as a number of dashes ('-'), one per letter in the word, so that you can see how many letters there are in the word, but not what the letters are. You will then guess a letter, one at a time. If the letter is in the word then the first occurrence in the word is revealed. If not, then a wrong guess is counted against you and that letter is banned from being tried again on this word. If you make too many wrong guesses then you lose.

Your program will also record the letters that you have tried and the order in which you tried them, as well as which letters are still valid to try.

### 1.2 Differences from the usual game

There are slight differences between the program that you will create here and the standard Hangman game. Firstly, (by default) you will only identify correct letters one letter at a time. e.g. if the word was 'Banana', the first guess of an 'a' would only identify the first 'a' in the word, the second would identify the second 'a', etc. Secondly, a letter is only removed as a possibility for guessing after it is guessed and NOT found in the word. So, the fourth try of 'a' for the word 'Banana' would fail, use up one of the wrong guesses, and mark the letter as no longer available for guessing.

In contrast, in standard Hangman, choosing a letter will reveal all occurrences of the letter in the word and will immediately make the letter unavailable for future guesses.

### 1.3 Purpose of the exercise

Adding all of these facilities to the program will mean that you will practice various different types of important C and procedural C++ programming concepts. The things that you learn, related to string manipulation and file input and output will be of use to you in the final coursework, so take this opportunity to learn them, get feedback in labs and find the coursework easier.

## 2 Deadline

There is no deadline. You can do this or not, it is up to you. I suggest that you do at least the parts related to memory management and strings though, and get feedback in the labs, because there are important skills to learn to pass the exam. The main purpose of this is to get you to learn these things.

## 3 Summary example of playing

A secret word is closen. Let us say that it is 'banana'. The letters in the secret word are only revealed to the player when they are correctly guessed. The table below shows an example of the guesses made (in the first column), the word which is displayed to the player (in the second column), the letters which can still be guessed (in the third column) and the letters guessed so far (in the fourth column).

| Guess | Displayed | Letters remaining | Letters guessed | Comment |
|---|---|---|---|---|
| | `------` | `ABCDEFGHIJKLMNOPQRSTUVWXYZ` | | Initial state |
| a | `-a----` | `ABCDEFGHIJKLMNOPQRSTUVWXYZ` | `A` | First A is revealed |
| A | `-a-a--` | `ABCDEFGHIJKLMNOPQRSTUVWXYZ` | `AA` | Second A is revealed |
| a | `-a-a-a` | `ABCDEFGHIJKLMNOPQRSTUVWXYZ` | `AAA` | Third A is revealed |
| a | `-a-a-a` | `-BCDEFGHIJKLMNOPQRSTUVWXYZ` | `AAAA` | No A. 1 wrong guess. |
| e | `-a-a-a` | `-BCD-FGHIJKLMNOPQRSTUVWXYZ` | `AAAAE` | No E. 2 wrong guesses. |
| C | `-a-a-a` | `-B-D-FGHIJKLMNOPQRSTUVWXYZ` | `AAAAEC` | No C. 3 wrong guesses. |
| b | `ba-a-a` | `-B-D-FGHIJKLMNOPQRSTUVWXYZ` | `AAAAECB` | First B is revealed |
| b | `ba-a-a` | `---D-FGHIJKLMNOPQRSTUVWXYZ` | `AAAAECBB` | No B. 4 wrong guesses |
| n | `bana-a` | `---D-FGHIJKLMNOPQRSTUVWXYZ` | `AAAAECBBN` | First N is revealed |
| t | `bana-a` | `---D-FGHIJKLMNOPQRS-UVWXYZ` | `AAAAECBBNT` | No T. 5 wrong guesses |
| N | `banana` | `---D-FGHIJKLMNOPQRSTUVWXYZ` | `AAAAECBBNTN` | second N is revealed |

At the end, the whole word has been revealed and the player wins. If the player has too many wrong guesses before they complete the word then they lose.

Note that the player can enter their letter as either lower case or upper case. It should be irrelevant. A lower case and upper case letter should match. Available letters were shown as upper case, but could be shown in lower case if you wish. That is up to you to choose. I displayed the current letters as lower case, but they could be shown as upper case if you wish, or as the same case as they are in the secret word. That is up to you to choose. I displayed the guessed letters here as upper case, but they could be shown as lower case if you wish, or as the same case as the letter typed in by the player. That is up to you to choose. In other words, you can display in whatever case you wish, but you need to ensure that whatever case the player enters the letter in does not matter (e.g. 'T' matches both 'T' or 't').

Letters are only removed from the available letters when a wrong guess is made. This is the only time at which the player can be sure that no more occurrences of the letter exist in the word.

Note: This version of hangman is much harder to win than the original version, because you will usually waste more guesses checking whether the letter occurs in the word again.

Please look at the examples document for more examples of play.

## 4 General requirements

This exercise has been divided into a number of different stages to aid you in working through and implementing the required functionality in a methodical way. You do not need to do the requirements in this order. Thus, you can follow the suggested steps if you wish, or just consider the requirements as a whole and develop a program to fulfill them all.

Your program should compile and run on the Computer Science Linux machines. You should test it by logging in to `bann` and compiling using `gcc`.

This is important! Always test your program on `bann`. (or another Computer Science Linux machine). Unlike Java, C and C++ are not totally portable, and differences between compilers can cause you problems.

## 5 Program requirements

There are a number of different elements to this exercise. Feel free to do as many or as few as you wish. I have written these out as requirements to give you some practice at implementing requirements, but you can ignore any of these or change them as you wish.

Basic requirements:

1. Handle command line arguments correctly.
2. Load a list of words.
3. Select a secret word.
4. Prompt the user for a guess, giving information about the current state (letters in the word and letters known not to be in the word).
5. Handle the guess that the user made.
6. Store all of the words and guesses, and display them to the user when prompted to do so.
7. Save the words and guesses to a file.

When you have a working program, test it using the sample input values provided in the examples document and verify that the output from your program is equivalent to that in the examples.

### 5.1 Command line arguments

Your program should accept a number of command line arguments. Multiple options can be supplied by the user and you must accept all, regardless of what order they appear in. Each configuration value is a single command line argument string, without spaces, so you can consider each independently.

You may want to get your program working without this flexibility first, then add the handling of command line arguments afterwards. However, since you saw command line arguments and string processing early on, it may actually be a good place to start.

The following command line arguments must be accepted:

- `-g<NUMBER>` or `-G<NUMBER>`

  Where `<NUMBER>` is a number, 0 or higher specifying the number of wrong guesses permitted.

This command line argument should specify how many wrong guesses are permitted before the player loses. For example, `-g12` will allow 12 wrong guesses, then the player will lose on the 13th wrong guess.

If this parameter is not specified then you should assume 10 wrong guesses are permitted.

- `-s<NUMBER>` or `-S<NUMBER>`

  Where `<NUMBER>` is a number, 0 or higher specifying the seed value.

  This command line argument is used to specify a seed number for the random number generator. If this parameter is not specified, then the random number generator should be seeded from the `time()` function. Note that the program should always display to the user the random number seed that was used, regardless of whether this was specified as input or the time was used.

  Note: Using the same random seed again in future (with the same dictionary) should generate the same word. This will be tested.

- `-f<FILENAME>` or `-F<FILENAME>`

  Where `<FILENAME>` is the path of a file containing the dictionary of words.

  Load the words from the file specified by the string filename.

  e.g. `-f./words.txt` to load from a file called `words.txt` in the current directory.

  e.g. `-F/mydir/words` to load from a file called `words` in the `/mydir` directory.

  If this option is not used then the `/lhome/jaa/words` file should be used as the dictionary.

  Note that we will test the program with a number of different files as well as the default file.

  i.e. all but the first 2 characters of the command line argument specify the file path of the words file to load.

- `-N` or `-n`

  Allow real names (assuming that the strings starting with capital letters are names). If this parameter is NOT specified then words which start with a capital letter should not be permitted. i.e. do not consider them in the word list. If this parameter IS specified then words which start with a capital letter should be considered.

Examples of use of command line arguments can be seen in the examples document.

## 5.2 Display a welcome message

You should display a message to the user specifying at least the following information:

- What the program is, e.g. "Welcome to CPP Hangman".
- What the name is of the file which contains the words which will be loaded/considered.
- What the random seed is that will be used.
- How many words were in the words file and what the length of the longest word is.

## 5.3 Choosing the word

A command line argument can be used to specify where the word list file is. If no filename is specified as a command line argument then you should load the word list which is in `/lhome/jaa/words` on `bann`. You do not need to load all of the words at once if you do not wish to do so, see below.

Note: Most Linux distros include a word list in the installation, but I couldn't find it on Bann (it is usually in somewhere like `/usr/share/dict`) so I put the words list from Marian into /lhome/jaa for you.

The words file which the program loads must have one word per line. i.e. words are separated by newline characters. (Depending upon how you read the file, you may need to remove the newline character from the end of the line which was read.)

Requirements:

- Load the dictionary file one line at a time. Count the number of valid words in the file.

  Note that you may store all of the words as you load them, or you may wish to just store the current word which was loaded.

  If you store just the one word at a time, then you will need to keep re-reading the file every time you want a new word, but this is probably easier for you to do, since you do not need to allocate memory to store a lot of words.

  If you store all of the words then you will need to make sure you allocate enough memory to store them all. You will see in a later lecture how to allocate memory dynamically. In that case you will not need to keep re-reading the file every time you need a new word. This will be trickier to do and more time consuming to debug so I have not made it a requirement. You may want to try creating an initial version which keeps rereading the file, then change it later when you understand about dynamic memory allocation.

- Whenever a word is needed, generate a random number (between 0 and the number of words minus one) to select a random word from the file (by reloading it until you get to the right word number) or from the list of words already loaded into memory.

- Load the word from the file (again) or the list of words in memory.

Once a word has been chosen, your program needs to continue to display the current state, accept user input, process the input, then display the new current state, etc, until the word has been guessed or the player makes too many wrong guesses. Your program then needs to choose another random word.

**IMPORTANT:** Words are only valid if they contain only alphabetic characters. Any word in the file which contains a non-alphabetic character (e.g. “1st”, “R&D” or “He’d”) should be rejected and not stored (or counted). This means that you need to check each character in the string to ensure that it is a valid letter (’a’ to ’z’ or ’A’ to ’Z’). You may want to leave this requirement until later, and get the program working without this feature first.

**Capital letters:** Note that words which begin with capital letters are probably real names. You can assume that names can be excluded from consideration by ignoring words which start with capital letters.

## 5.4 Display the current state

The current state consists of the following:

- The word that is being guessed. Letters which have not yet been correctly guessed should be shown as a dash (−). Letters which have been guessed should be shown.

- The letters that can still be used. If a letter is chosen and is not found in the word then the letter is removed from the list of letters which are still available.

- The letters that have been tried so far, in the order in which they were tried. This includes both correct and incorrect guesses.

The player should be shown the current state of the secret word, the valid guess letters and the letters guessed so far. The player should also be told how many wrong guesses they are still permitted. The player should then be prompted for their new guess.

The word which is shown to the players will always have the same number of characters as the secret word (each character is a dash or the correct letter).

There will always be 26 characters displayed to the player for the available letters. Each character will be a dash (‘-’) for an unavailable letter, or the letter if it is available.

The number of letters tried so far will initially be zero, then letters will be added as they are tried. Note that the maximum number of guesses that need to be stored is equal to the number of letters in the word plus the number of wrong guesses permitted. (At that point either they player would have guessed the word or have had too many wrong guesses.) This information may be useful to you when deciding upon how much memory to allocate.

## 5.5 Allow user to guess a letter

The program should accept a single character from the user (followed by ENTER or RETURN). If multiple characters are provided then the first should be accepted and all others up to the ENTER/RETURN should be thrown away.

- If the user specifies an upper case letter, it should be converted into lower case.
- If the user specifies an upper or lower case letter that is still available, then this is a guess and should be handled as specified below (see “Matching a guess”).
- If the user specifies a letter which is not currently available then tell the user, ignore the input and prompt the user again for another letter.
- The following special characters (`! ? * $`) should also be accepted. If the user enters one of these then it is NOT a guess (i.e. do not record it in the list of guesses). Instead handle the character in the following way:

|   |   |
|---|---|
| `!` | Exit the program now. |
| `?` | Cheat. Display the current secret word to the user. |
| `*` | Display the words that have been used so far, along with the guesses. See below. |
| `$` | Save the current words and guesses to a file. See below. |

### 5.5.1 Matching a guess

When a guess is made, your program has to check whether it is correct or not. Look through the letters of the current word which have not yet been found and find the first occurrence of the letter in the word. **IMPORTANT:** Checks should be case-insensitive. i.e. “a” should match an “a” or an “A” when checking for letters in the word.

If an unidentified copy of the letter is found in the word (i.e. the letter is in the word and has not already been revealed), then reveal the letter for the first place in which it was found and check whether the whole word has been identified. If the whole word has now been identified (i.e. all letters have been revealed), then tell the player that they succeeded, choose another word and continue from the start. (See sections 5.6 and 5.7 below - you need to keep a record of the old word and the guesses.)

Otherwise, if the letter is not in the word, remove the letter from the available letters list and increase the count of the number of wrong guesses made so far. If the player has made too many wrong guesses then they lose. In that case, tell the player that they lost, tell them the word, choose another word and start from the beginning again. (Note: letters are ONLY removed from the available letters list when they are not found in the word.)

Regardless of whether the guess was correct or not, add the guess letter to the list of letters which have been tried - in the order in which they were tried.

If this is not clear, then please look at the examples document.

## 5.6 Store and display words and guesses

**Note:** You will probably want to leave this requirement until later, and do the basic requirements first. In particular, you will want to have seen the lectures on dynamic memory allocation and linked lists before you try to do this.

You should store all of the words that have been been chosen so far, along with the guesses that were made. Whatever storage method you choose, you should not limit the number of words that can be stored (e.g. use a linked list or an array which you resize when needed). You should justify your approach (e.g. linked list, array, or other, and why you used it) in your `readme.txt` file. When the user chooses the `*` option, you should display a list of the words which have been chosen so far, and all of the guesses, in the order in which the words were chosen.

For each secret word your program should display:

- The word number, e.g. 1 for the first, 2 for the second, etc.
- The word itself.
- The final state of the guesses made, or the current state. i.e. the correct letters found and '`-`' for the letters which were not guessed.
- The letters remaining.
- The letters which were guessed, in the order in which they were guessed.

Example:

```
Word number : 1
Word        : 'grub'
Final guess : '--u-'
Remaining   : '-BCD-FGH-JKLMN-PQRSTUVWXYZ'
Guesses     : 'AEIOU'
```

## 5.7 Save the words and guesses

**Note:** You could make a start on this requirement by saving just the current word and guessed letters. You will need to implement the ''store' requirement before you can fully implement this requirement.

If the `$` option is chosen, then you must save the current words and guesses. You should ask the player for a filename. After getting the filename, display it back to the player and ask for confirmation. If confirmation is given then save the words and guesses to the file. This involves opening/creating the file, writing the list of words and the guesses into it and closing the file.

Your file must include the secret word, the guesses that were made and the letters in the word that had been guessed. You should put each secret word, the identified letters, the letters remaining and the guessed letters on a single line for each secret word.

For example, the following is a possible content for a saved words/guesses file:

```
Word 0: 'gaslight', 'gaslight', '-BCD-FGHIJKLMN-PQRST-VWXYZ', 'AAEIOULGSGHT'
Word 1: 'fourteenth', 'fourteenth', '---D-FGH-JKLMN-PQR-T-VWX-Z', 'ABEIOUTEEOUSYCTHRN
Word 2: 'curlicue', 'cu-li-ue', '-BC-EF-H-JKL----QR--UV-XYZ', 'AEIOUUISTGWNMDCLP'
Word 3: 'genera', '------', 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', ''
```

## 6 Documentation and help

You can obtain help from at least four places:

1. Use the UNIX manual pages. Type `man` followed by the name of the function which you need to look up. This is the way I'd advise since it will help you in future.
2. Run up your favourite search engine in your web browser and type the function name with the word C (so you get C language help).

   e.g. `printf c`

   Then select from the many web pages which will be shown.
3. If you are using the Windows PCs in A32: Open the source code file in **Visual Studio**, select the keyword or function name which you want help on and press F1.
4. Ask at the organised lab sessions. Options 1-3 will help to better prepare you for the future though.

## 7 Change history:

Version 1.0: Initial version.