# G52CPP, 2017/18, Coursework Part 1, Hangman

**Submission: Ideally have this marked in the labs by 12th February. Deadline is actually the week after (19/2/2018) to avoid you getting late marks, but if you miss the 12/2/2018 date you will be getting behind. Submit the code to moodle as well for a permanent record in case of queries.**
**No late submissions/marking beyond the 19/2/2018 without extenuating circumstances.**

## Overview

Hangman is a word guessing game. You guess the word one letter at a time. When you correctly guess a letter all matching letters in the word are shown. You are only allowed a limited number of wrong guesses before you lose. (Traditionally, every wrong guess meant that a part of the hangman was drawn, but we won't worry about that – we'll just keep a count of allowed wrong guesses.)

Using C++ and the standard C library functions, you will create a simplified Hangman. I have provided an example version in a windows executable file which you can try out to ensure that you understand the aims of the coursework.

You should write your program and then submit your source code to moodle and demo your program in the lab to get it marked (for instant feedback). You can demo your program on any operating system you wish (e.g. Windows, Mac, Linux) using a standard C++ compiler (any version), but your code should be standard C/C++ code which will compile on standard C/C++ compilers. The moodle submission is needed to ensure that there is a permanent copy of your coursework in case of any queries or issues with marking. In general as long as you have it marked in the lab your mark will be recorded.

Note that coursework part 2 is very similar to this and builds upon part 1 (you will be altering the code for part 1), so it is worth doing this coursework correctly. Together parts 1 and 2 are 12% of the module.

## Requirements

Your program should work similarly to the example program provided, which you can download and run to see how it should work. The following 6 requirements are worth 1% of the module mark each. *You need to be prepared to explain your work to the markers when necessary in order to get the mark.*

1) Seed the random number generator at the start, and use the random number function to pick a random word each time. The program should also be runnable with no command line (in which case seed the random number generator with the time()) or with a single number as a command line argument (in which case seed the random number generator with that number). [Requires you to know something about the command line arguments, rand(), srand(), time(), atoi() or similar functions.]
2) Create an array of string literals to store a set of hard-coded words to use for the secret words. Initialise the array with your set of string literals when you do so. Create strings (char*s/char arrays) for the available characters (will be a fixed length) and letters in the current word (will need to determine the length dynamically, depending on the current word. You should have at least 5 words in your array. [Requires knowledge of simple arrays, strings and initialisation.]
3) Use malloc() to allocate the memory for the current word guess and remember to free() it when it is not needed. You can test your system without having to do this requirement by just using a big array of characters (bit enough for the longest word), but you will need to use malloc() and free for the final version. [Requires you to demonstrate used of a simple malloc() and free().]

4) Display an appropriate prompt to the user, including the current word (initially a set of '----', later with filled in letters, e.g. 'he-p'), the letters which have not been guessed (initially the letters 'abcdefghijklmnopqrstuvwxyz', then remove any letters which are guessed over time, e.g. by replacing with a . or -, as in 'abcd.fg.ijklmno.qrstuvwxyz'), and the number of wrong guesses still permitted. [Requires some text output, including strings.]
5) Accept an input character and handle it correctly, showing an appropriate message to the user depending upon whether the letter is in the word, not in the word, or already guessed (correctly determining which of these is the case for the specified letter). When a letter is guessed, modify the string of available letters to remove it, and either mark it in the word (e.g. change '----' for the word 'help' to '---p' when the letter p is guessed) for a correct guess, or decrease the wrong guesses for an incorrect guess. Mark off the letter from the list of available letters. [Requires character input and string (char array) modification.]
6) Tell the player when they fail (run out of guesses) or succeed (guess every letter) and start again correctly with a new word. [Requires you to detect this and to be able to reset or recreate all variables appropriately. Ensure that all 'malloc'ed memory is 'free'd.]

**Additional marking considerations/requirements – these may modify the mark:**

- Use only standard library functions, not additional custom libraries (compulsory to get any marks)
- Use the C++ style for include files, e.g. <cstdio> not <stdio.h> (compulsory to get any marks)
- Make values const where appropriate. (May lower marks if you do not.)
- Ensure that the checks are case-insensitive. The easiest way is to change both the letter to compare against and the letter typed into lower case before comparing them. (May lower marks if not.)
- Ensure that you free() all memory that you malloc().You do need to malloc() at least one thing. (WILL lower marks if you do not free the memory.)
- Ensure that your program does not crash and works well before you demo it. (Crashes and errors will lower marks significantly.)

**Note on Visual Studio:**

I wrote the program inside visual studio, so needed the following line at the top of the file to turn off the warnings about using risky functions like strcpy():
```
#define _CRT_SECURE_NO_WARNINGS
```

## Marking criteria

This coursework is worth 6% of the module mark. You should be able to get at least most of it done in the two lab sessions but I suggest working on it outside of labs and coming to the labs with the questions when you get stuck.

There are 6 criteria above (for 6 marks, 1% each), so you should ensure that when you demo your program you can demonstrate that all 6 criteria were met in order to get the marks for the coursework.

The additional considerations need to be considered since they can cause you to lose some or all of the marks! (You should be able to handle all of these relatively easily though.)

As long as you meet the criteria above you can structure your program in any way you wish. Feel free the experiment – the primary aim is to ensure you are happy with strings and pointers really. We really will check that you met the wording of the six marking criteria and that your program actually works, so everything else which is not mentioned on this and the previous page can be considered to be flexible.

## The rest of this document is just help and hints, so feel free to ignore it if you know already what you are doing.

### How to get started:

- Try the example program. Ensure that you know what it is doing.
- Read the requirements and plan out how you will create a program to meet them. The aim of the requirements is to help you not to miss something important.
- Try the three demos demo1a, demo1b, demo1c from the demo/practical lecture. These tell you how to code up most of the facilities – especially arrays of strings, command line arguments and random numbers.
- If you are stuck, see my notes on the next couple of pages about my answer. It should give you some clues if you get stuck. Note however that you could do the coursework in a completely different way to me, in which case the information may not match what you do – that is not a problem.
- You may want to wait until after Lecture 3 to work on this, then start it in the lab, but I expect you should know enough to get started before then from the first year work and lecture 2/tutorial.

### Hints from my example answer for this coursework

I just used one function – everything was inside the main. You'll probably find it better to use more functions though and it may make the code clearer. Either will not affect your mark.

The length of your program will not affect your mark. To give you an idea of the sort of size of the coursework, my program was 99 lines long but had 10 blank lines in that – separating parts, 5 lines were #includes, two lines were comments only (no code), and 18 lines consisted of only a single open or close brace and 58 actually had code on them. This means that you don't need a huge program. Note that I could have made it a lot shorter by putting multiple statements on a line, but wanted it to be more readable. I also put one variable declaration per line, so that makes it seem bigger than necessary.

#### Standard library functions I used

I used the following functions: srand(), rand(), strlen(), malloc() (for the letters guessed so far), memset(), free(), printf(), getchar(), tolower(), strcpy(). You can use other standard library functions if you wish, but I mention these because it hopefully gives you hints – I suggest to know what all of these do.

#### Examples of variables you may need

I used the following local variables:

- **arrWordList** (array of words)
- **iWordCount** (count of words in the array – hardcode it or calculate from array size)
- **iTargetWordNumber** (index of currently selected word – i.e. random number from 0 to (iWordCount-1))
- **iWordLength** (length of currently selected word – since this gets used a lot)
- **arrLetters** (array of all of the 26 letters – you don't need this and could use a string literal)

- **arrAvailable** (array of 26 letters where I replace them by a . when used – I initialised this with a strcpy from ‘arrLetters’ but you don’t need to do so )
- **pCurrentWord** (currently selected word – not really needed but easier than using the array index all of the time)
- **pBlankWord** (a copy of the word which is initially all ‘-‘ characters, but gets filled in as letters are guessed – this is the only memory that I needed to malloc since I didn’t know how long it was initially until the word is chosen)
- **iLettersGuessed** (number of letters guessed so far)
- **iWrongGuessesLeft** (number of wrong guesses so far)

You can use whatever variables you wish, but sometimes it’s hard to get started so I thought that giving you some ideas of the type of data you need to store may help some of you.

## Example of program structure, if you need ideas

If you are stuck on getting started, the simplified pseudocode below shows a simple structure for the program that I created. You are welcome to follow this structure or to create your own structure. i.e. this is an example that you can feel free to ignore.

Note: Coursework 2 will build on this, extending it and adding some more C++ concepts so it is worth making sure that you do this and understand it now, to avoid issues later. Coursework parts 1 and 2 fit together. Coursework parts 3 and 4 also fit together to do something very different.

The structure of my answer program can be summarised as follows:

```
Check the command line argument and seed the random number with either the time (if no
command line argument) or the command line argument if there is one.
Initialise the things like the words array and size of array
Loop forever (I did a ‘return’ when player says don’t try another word)
    Choose a random word and set up the current word, available letters (to all letters, i.e.
    “abcd…”) and letters found so far (to all ‘-‘, i.e. ‘----…’)
    Loop until all letters have been guessed, or player runs out of guesses
        Tell player current state – letters in word, letters left, etc
        Ask for a letter from the player. If/while it’s not valid then ask for another one
        If letter already guessed then tell player and do nothing
        Else if letter is in the word then mark the letters identified, count of letters found,
        and tell player
        Else letter is wrong so decrease the number of wrong guesses left
    End of loop until all letters guessed or player lost
    Inform player of winning or losing and ask whether they want to have another game. If so
    then allow the infinite loop to repeat. If not then return from main().
End of infinite loop
```

I note that you may find the program structure a lot easier if you use functional decomposition to split the program into multiple functions. This is good, but I wanted just to illustrate that it’s not vital to do so.