Hangman

COLLABORATORS

	TITLE : Hangman		
ACTION	NAME	DATE	SIGNATURE
WRITTEN BY	Jordi Fita	February 6, 2018	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
9bd2d2b87442	2011-08-01	Move hangman's links as image to the top.	jfita
02966c43ff92	2011-04-14	Removed the laying bit about rand() mod x not being normal.	jfita
34b7522b4f97	2011-03-28	atangle is now using a new style for directives which don't collide with XML tags. I had to update all games and programs as well in order to use the new directive syntax.	jfita
694b0ad0d656	2011-03-25	Fixed the link to the normal distribution in hangman.	jfita
6cc909c0b61d	2011-03-07	Added the comments section.	jfita
3c0dd7be5c1d	2010-10-28	Corrected URL to atangle.	jfita
d5cc1bcb5948	2010-10-28	The correct source language for Makefiles is make.	jfita
854feca7a1b6	2010-10-27	Added the source style and thus highlighting to the Makefile.	jfita
93aeeadf8222	2010-10-26	Added the screenshot of hangman.	jfita
78645501eebf	2010-10-25	Added the Makefile to hangman.	jfita

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
c00f903348be	2010-10-25	Added AsciiDoc's homepage's link to hangman.	jfita
05a1b32f8b4a	2010-10-22	The appendix sections now aren't actual appendix when making a book.	jfita
6bd3e013eaf8	2010-10-22	Added the downloads to hangman.	jfita
c3f41549a137	2010-10-22	Added the check for eof on stdin when reading letters.	jfita
e81e67555c84	2010-10-22	Reworded some sections that weren't clear enough.	jfita
7207caac3857	2010-10-21	Added the initial version of hangman.	jfita

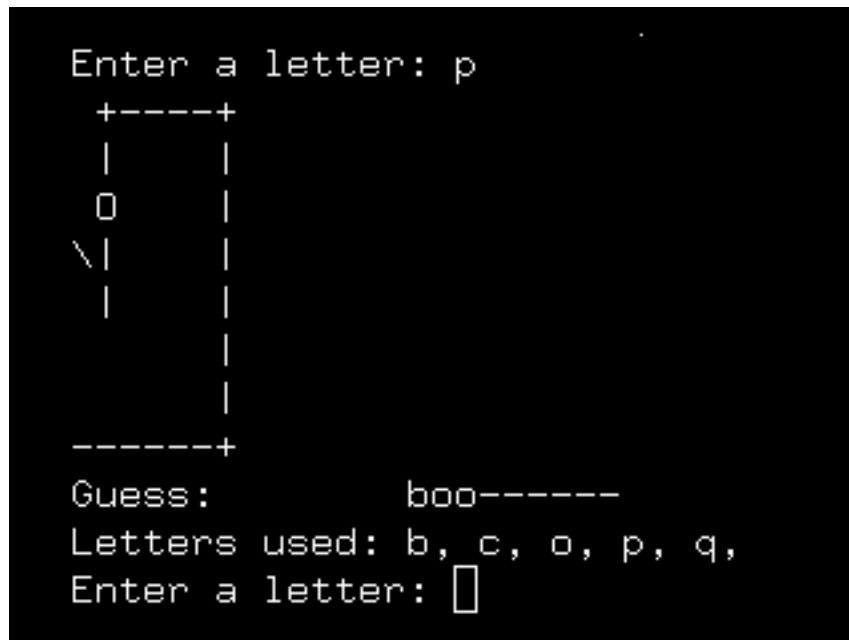# **Contents**

List of Figures

1 Introduction

Figure 1: Screenshot of hangman with some letters guessed.

Hangman is a simple guessing game in which the player tries to guess a word by suggesting letters until she either finds out the word or has attempted to guess too many times.

This version of `hangman` is a simple C++ command line application that randomly selects a word from a file and then ask the player to guess letters. The word to guess is represented on screen by a row of dashes (−), one for each letter still to be guessed. Each time the player guesses a letter correctly, the computer will show the letter in the correct position instead of the dash. If the guess is incorrect, then it will draw a piece of a hangman diagram.

2 The Main Logic

The application will start off building a list of words from a file and then will pick up a random word from that list. Then it will ask the player to enter a letter that hasn't been used already. It will check whether the word contains the entered letter and depending on that will update the screen replacing dashes with the letter or draw another piece of the hangman's diagram.

The game ends when the player either guesses the word correctly or the hangman diagram is complete. At this time, the application will ask whether to play again with a newly picked word or quit.

Being written in C++ the application can throw exceptions when there's something wrong, therefore the first thing to do is place a `try catch` block that catches any exception in order to show the error message and avoid calling `abort()` due to an exception leaving `main`.

```
<<main>>=
int main()
{
    try {
```

```
        <<hangman code>>

        return EXIT_SUCCESS;
    } catch (std::exception &e) {
        std::cerr << e.what() << std::endl;
    } catch (...) {
        std::cerr << "Unknown error" << std::endl;
    }

    return EXIT_FAILURE;
}
```

I use the `EXIT_SUCCESS` and `EXIT_FAILURE` macros defined in C's standard library.

```
<<includes>>=
#include <cstdlib>
```

I `catch` all thrown exception objects that derive from the standard's `exception` class as well as any other object the application might throw. In the latter case, I just print out an unknown error. If there is no exception thrown, then the application hits the `return` statement inside the `try` block and exits with `EXIT_SUCCESS`.

For this block to work, I need to include the header with the definition of the standard exception classes and the `cerr` object.

```
<<includes>>=
#include <stdexcept>
#include <iostream>
```

Inside the `try` block, then, I'll place the main application's code.

```
<<hangman code>>=
bool quit = false;
<<build word list>>
<<initialize random number generator>>
do {
    <<pick random word>>
    <<build guess>>
    unsigned int attempts = 0;
    do {
        <<draw current status>>
        <<ask for new letter>>
        <<check whether the letter belongs to the word>>
    } while (guess != word && attempts <= 6);
    <<check whether the player won or lost>>
    <<draw current status>>
    std::cout << "The word was: " << word << std::endl;
    <<ask whether play again>>
} while (!quit);
```

3 Building the List of Words

The application will build the list of words to use to play from a simple plain text file. This file is named `hangmanwords.txt` and is located at the same working directory as the application. If it can't open the file, it can't continue and throw an exception. The more appropriate exception in this case is `runtime_error`, but I don't need a derived class because I don't need to add additional information about the error and also don't need to treat this error specifically.

```
<<open the words file>>=
std::fstream wordsfile("hangmanwords.txt", std::fstream::in);
if (!wordsfile) {
    throw std::runtime_error("Couldn't open words file");
}
```

To use `fstream` I need to add the necessary header file.

```
<<includes>>=
#include <fstream>
```

Even though I could just read the file line by line to know how many words there are, in an effort to improve efficiency the first line will contain the number of words in the file.

```
<<hangmanwords.txt>>=
5
```

This number is just a *hint* for the application to reserve the memory necessary to hold *at least* as many words as specified. Given that I don't need to add nor remove words once they have been read from the file, and since I will access the list in random order, the best structure in which to store the list of words is to use a `vector` of `string`. So, after clearing the vector, `reserve` as many elements as specified in the file.

```
<<reserve space for words>>=
int numberOfWords = 1; // At least I'll have a single word.
wordsfile >> numberOfWords;

wordlist.clear();
wordlist.reserve(numberOfWords);
```

To keep things simple, each word will be in a line of its own and will not have heading nor trailing spaces. The word can't also have spaces.

```
<<hangmanwords.txt>>=
unicorn
hangman
boogieman
cromulent
supercalifragilisticexpialidocious
```

Therefore I need to read the file line by line and store every non-empty word to the `vector`. To avoid a potential buffer overflow, I'll use `getline` defined inside the `string` header file.

```
<<includes>>=
#include <string>
```

`getline` reads a whole line from an input stream and stores the contents to a `string` object. I'll keep reading words and storing them inside the `vector` until I've read the entire file. I must check if the word is empty before adding it to the list to screen out blank lines.

```
<<read words>>=
while (!wordsfile.eof()) {
    std::string word;
    getline(wordsfile, word);
    if (!word.empty()) {
        wordlist.push_back(word);
    }
}
```

If there is no word in the list, I can't continue and so throw another exception. Again, given no need to add additional information nor handle this error in any special way, using a simple `runtime_error` is fine.

```
<<check whether the word list is empty>>=
if (wordlist.empty()) {
    throw std::runtime_error("No words in the list");
}
```

Finally, as the word list could be big, instead of returning the `vector` as the function's result value, the function asks for a reference to an already allocated `vector` object in which it will fill up with the words. Once the function return, the words will be stored in the referenced `vector`.

```
<<build word list function>>=
void
buildWordList(std::vector<std::string> &wordlist) {
    <<open the words file>>
    <<reserve space for words>>
    <<read words>>
    <<check whether the word list is empty>>
}
```

```
<<build word list>>=
std::vector<std::string> wordlist;
buildWordList(wordlist);
```

To use `vector` I need to include the appropriate header.

```
<<includes>>=
#include <vector>
```

4 Picking a Random Word

To pick up a random word from the word list, I'll use C's standard `rand` function. This function returns a number between 0 and `MAX_RAND`, which is defined inside the `cstdlib` header. Before using this function, though, I must initialize the random number generator using a *seed*. This seed needs to be a different number each time the application starts. A good seed in this case is to use the current time in which the application starts.

```
<<initialize random number generator>>=
srand(time(0));
```

The `time` function is defined inside the `ctime` header.

```
<<includes>>=
#include <ctime>
```

When the random number generator is initialized, I need to get an index inside the `wordlist` vector to select a word to guess. I'll use the output of the `rand` function limiting its range to the word list's length using the modulo.

```
<<pick random word>>=
std::string word = wordlist[rand() % wordlist.size()];
```

5 Guessing

The application need to keep an string with the same length as the word to guess but with every letter replaced with a dash. The easiest way is to use the `string` constructor that accepts a length and a character to fill the string with.

```
<<build guess>>=
std::string guess(word.size(), '-');
```

I also need to keep score of the letters the player tried to prevent the player to use the same letter twice. I'll just add the letters tried to a `set` of `char`, which doesn't allow duplicates.

```
<<includes>>=
#include <set>
```

```
<<build guess>>=
std::set<std::string::value_type> lettersUsed;
```

5.1   The Current Guess

The application should show to the player the current game's status. The status is the guess word with the already guessed letters displayed while the other remain shown as dashes, and the drawing of the hangman diagram.

```
<<draw current status>>=
drawHangman(std::cout, attempts);
std::cout << "Guess:        " << guess << "\n";
```

The function that draws the hangman requires the number of attempts and based on that draws a part or the whole hangman diagram.

```
<<draw hangman function>>=
// ASCII ART FTW!
void drawHangman(std::ostream &out, unsigned int attempts)
{
    switch(attempts)
    {
        case 0:
            out << " +----+\n";
            out << " |    |\n";
            out << "      |\n";
            out << "      |\n";
            out << "      |\n";
            out << "      |\n";
            out << "      |\n";
            out << "------+\n";
            break;

        case 1:
            out << " +----+\n";
            out << " |    |\n";
            out << " O    |\n";
            out << "      |\n";
            out << "      |\n";
            out << "      |\n";
            out << "      |\n";
            out << "------+\n";
            break;

        case 2:
            out << " +----+\n";
            out << " |    |\n";
            out << " O    |\n";
            out << " |    |\n";
            out << " |    |\n";
            out << "      |\n";
            out << "      |\n";
            out << "------+\n";
            break;

        case 3:
            out << " +----+\n";
            out << " |    |\n";
            out << " O    |\n";
            out << "\\|    |\n";
            out << " |    |\n";
            out << "      |\n";
            out << "      |\n";
            out << "------+\n";
            break;
```

```
        case 4:
            out << " +----+\n";
            out << " |    |\n";
            out << " O    |\n";
            out << "\\|/   |\n";
            out << " |    |\n";
            out << "      |\n";
            out << "      |\n";
            out << "------+\n";
            break;

        case 5:
            out << " +----+\n";
            out << " |    |\n";
            out << " O    |\n";
            out << "\\|/   |\n";
            out << " |    |\n";
            out << "/     |\n";
            out << "      |\n";
            out << "------+\n";
            break;

        default:
            out << " +----+\n";
            out << " |    |\n";
            out << " O    |\n";
            out << "\\|/   |\n";
            out << " |    |\n";
            out << "/ \\   |\n";
            out << "      |\n";
            out << "------+\n";
            break;
    }
}
```

It is also nice to show the already used letters. For that, I'll copy the set's contents to just draw to an `ostream_iterator` to output to `cout` and add a comma between the letters in the `set`.

```
<<includes>>=
#include <algorithm> // for copy
#include <iterator> // for ostream_iterator
```

```
<<draw current status>>=
std::cout << "Letters used: ";
std::copy(lettersUsed.begin(), lettersUsed.end(),
    std::ostream_iterator<std::string::value_type>(std::cout, ", "));
std::cout << "\n";
```

5.2 Guessing Letters

The application must ask for new letters to the player. Wait and read a letter from the standard input and check if the entered letter is in the `set` of used letters. Ask again for a new letter until the entered letter is not inside the `set`. Once the player enter new letter, add it to the set.

Take notice that I need to check for the standar input's end of file, because the player could close the input (CTRL+Z or CTRL+D) and in that case the game would start an infinite loop asking for a letter from an stream it can't read from. If the application detects this, it assumes the player wants to quit.

```
<<ask for new letter>>=
char letter = '\0';
```

```
do {
    std::cout << "Enter a letter: ";
    std::cin >> letter;
} while (lettersUsed.find(letter) != lettersUsed.end() && !std::cin.eof());
// We can't read anymore with a closed stdin.  Quit.
if (std::cin.eof()) {
    return EXIT_SUCCESS;
}
lettersUsed.insert(letter);
```

Then I check whether the word has the entered letter. Here, the simplest and cleanest way is to iterate the word and compare each string's character with the letter entered. For each letter that I find in the word, I replace the dash in the guess word with the actual letter. After that, check if the guess word has changed and increment the number of attempts if it didn't, because it means that the word has not that letter.

```
<<check whether the letter belongs to the word>>=
bool guessWordChanged = false;
for (size_t char_pos = 0 ; char_pos < word.size() ; ++char_pos) {
    if (letter == word[char_pos]) {
        guess[char_pos] = letter;
        guessWordChanged = true;
    }
}
// no luck
if (!guessWordChanged) {
    ++attempts;
}
```

5.3 Does the Winner Play Again?

Once the player guessed the correct word or guessed to many letters incorrectly, the application prints out whether the player won or not.

```
<<check whether the player won or lost>>=
std::cout << "\n\n";
if (guess == word) {
    std::cout << "YOU FOUND THE WORD!\n";
} else {
    std::cout << "You didn't find the word.  Good luck next time.\n";
}
std::cout << "\n";
```

The last thing it needs to do is ask to the player if she wants to play another round. If she enters a `y` then application starts again picking a new word and ask for letters. With any other character, it quits.

```
<<ask whether play again>>=
std::cout << "Play again (y/n)? ";
char playAgain = 'n';
std::cin >> playAgain;
std::cout << std::endl;
quit = playAgain != 'y';
```

Here I don't need to check for `eof` on `stdin` because in this case the read wouldn't do anything, keeping the `playAgain` variable to `n` and thus would quit anyway.

A   hangman.c

A single source code module holds all the building blocks I've described before.

```
<<*>>=
/*
   Hangman - A simple console hang man game.
   Copyright (c) 2010 Jordi Fita <jfita@geishastudios.com>

   <<license>>
*/
<<includes>>

<<build word list function>>

<<draw hangman function>>

<<main>>
```

B Makefile

Being a simple application, an small Makefile would be sufficient to build and link `hangman` from the source document.

The first thing that needs to be done is to extract the C++ source code from the AsciiDoc document using `atangle`. It is necessary, therefore, to have a `atangle` installed to extract the source code.

```
<<extract cpp source code>>=
hangman.cpp: hangman.txt
        atangle $< > $@
```

It is also possible to extract a sample word list from the same AsciiDoc document by using a different root node, in this case `hangmanwords.txt`. This word list is just a sample and only contains five words, but it useful to make the application run and it can be easily extended using any text editor.

```
<<extract word list>>=
hangmanwords.txt: hangman.txt
        atangle -r $@ $< > $@
```

Then is possible to link the executable from the extracted C++ source code. Although, I have to take into account the platform executable suffix. For Linux and other UNIX systems, the suffix is the empty string, but for Windows I need to append *.exe* to the executable.

To know which system is the executable being build, I'll use the `uname -s` command, available both in Linux and also in MinGW or Cygwin for Windows. In this case, I only detect the presence of MinGW because I don't want to add yet another dependency to Cygwin's DLL.

```
<<determine executable suffix>>=
UNAME = $(shell uname -s)
MINGW = $(findstring MINGW32, $(UNAME))
```

Later, I just need to check if the substring *MINGW* is contained in the output of `uname`. If the `findstring` call's result is the empty string, then we assume we are building in a platform that doesn't have executable suffix.

```
<<determine executable suffix>>=
ifneq ($(MINGW),)
EXE := .exe
endif
```

With this suffix, I can now build the final executable. Notice how besides the C++ source code I also make the word list a dependence. By this, I make sure that the application can run just after linking without any extra step.

```
<<build hangman executable>>=
hangman$(EXE): hangman.cpp hangmanwords.txt
        g++ -o $@ $<
```

Sometimes, it is necessary to remove the executable as well as the intermediary building artifacts. For this, I'll add a target named clean that will build all the files built by the Makefile and only left the original document. I have to mark this target as PHONY in case there is a file named clean in the same directory as the Makefile.

```
<<clean build artifacts>>=
.PHONY: clean

clean:
        rm -f hangman$(EXE) hangman.cpp hangmanwords.txt
```

As the first defined target is the Makefile's default target, I'll place the executable first and then all the dependences, until the original document. After all source code targets, I'll put the clean target. This is not required, but a personal choice. The final Makefile's structure is thus the following.

```
<<Makefile>>=
<<determine executable suffix>>

<<build hangman executable>>

<<extract cpp source code>>

<<extract word list>>

<<clean build artifacts>>
```

C License

This program is distributed under the terms of the GNU General Public License (GPL) version 2.0 as follows:

```
<<license>>=
This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License version 2.0 as
published by the Free Software Foundation.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307  USA
```