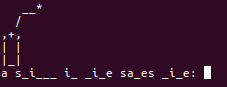
Raising Your Skill – Boom

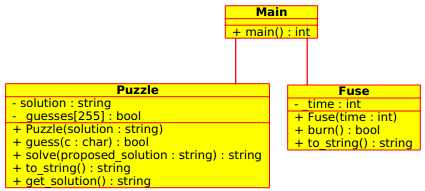
CSE 1325-004 – Fall 2017 – Homework #3

Due Thursday, September 14th at 11:59 pm

This assignment moves us past simple single-file programs into more interesting multi-file programs. We also write a game that, if not a viable League of Legends™ competitor, is at least reasonably enjoyable at the Extreme Bonus level. Along the way, we’ll utilize a Makefile, either the ddd or gdb debugger, and Umbrello to design our class and use case diagrams.

# Full Credit Requirements

Boom is a classic word or phrase guessing game with limited tries and something catastrophic if the word is missed – a firecracker goes (you guessed it) “Boom!”. A game in progress looks something like the screen shot to the right.

Below is a simple UML class diagram of the Full Credit version of the game. Note that Main isn't actually a class – it’s simply the usual main function. This time around, implement a .h and .cpp for each class.

The Puzzle class represents the game. The \_solution field is the string that is the word or phrase the player is trying to guess. The \_guesses array (like a vector, but without methods) holds a Boolean for each ASCII character. For example, guesses[‘a’] is true if ‘a’ has been guessed, false if not.

The Puzzle constructor accepts the solution, which should be stored in \_solution. The guess method accepts a single character (the player’s guess), and returns true if the character is valid (between ‘a’ and ‘z’, and not yet guessed) and false otherwise. It should also update \_guesses. The solve method accepts a proposed solution, and returns true if correct (it matches \_solution) or false if incorrect. The to\_string method returns the player’s view of the puzzle – that is, characters that have been guessed are visible, those that have not been guessed are replaced with a ‘\_’, and spaces are not changed. The get\_solution getter method simply returns \_solution.

The Fuse constructor accepts the time (i.e., number of guesses) that may elapse before the firecracker goes boom. The burn method decrements the remaining time, and returns true if any time remains or false if time as expired. The to\_string method returns an ASCII art representation of the firecracker, with the number of segments of fuse representing the time (aka number of guesses) remaining.

Main will implement the following algorithm:

(1) Create the variables.

(2) Display a welcome message, like this:

=================

B O O M !

=================

Enter lower case letters to guess,

! to propose a solution,

0 to exit.

(3) Begin the main loop:

(3a) Display the firecracker and the player’s view of the puzzle, with a prompt, something like this:

\_\_\_\_\_\_\_\_\*

/

,+,

| |

|\_|

\_ \_\_\_\_\_\_ \_\_ \_\_\_\_ \_\_\_\_\_ \_\_\_\_:

(3b) Accept a character from the player. If it’s a 0, immediately exit. If it’s an exclamation point (!), ask for a proposed solution: If correct, they win, if not, they lose. If the character is an invalid guess, print “Invalid character - try again”. Otherwise, update the state of the puzzle using the player’s guess, including decrementing the length of the fuse.

(4) After a proposed solution, or if the firecracker explodes, display something like this if the player won:

\*\*\* W I N N E R \*\*\*

and something like this if the player lost:

###### BOOM ######

The answer was: haste makes waste

At this level, it is acceptable to hard-code the solution string – not much of a challenge for replaying, but it’s a start.

The student will create a simple Use Case diagram in Umbrello for the game with one actor (Player) and three use cases: Guess a Letter, Guess the Solution, and Exit. No additional documentation (diagrams or text) is required for each use case. We will go over how to do this in Umbrello on Monday.

As with previous homework, the student will deliver a ZIP archive file with a Full Credit subdirectory. The zip file should be called abc1234\_HW3.zip

For credit, the full credit subdirectory will contain the following source code files corresponding to the class diagram: abc1234\_main.cpp, abc1234\_puzzle.h, puzzle.cpp, abc1234\_fuse.h, and abc1234\_fuse.cpp. The student will also deliver in this subdirectory a correct Makefile, a abc1234\_puzzle.png image file demonstrating the game in progress, and **an abc1234\_puzzle-usecase.png image of the use case diagram.**

# Bonus 1 Requirements (10 pts)

If you played a similar game in school, you probably didn’t use one of your fuse links if the character you guessed was part of the word.

At this level, Puzzle::guess will be modified such that returning true indicates the guessed char was indeed in the solution, while returning false indicates that it was not (and thus the fuse on the firecracker should decrement). Define an exception Bad\_char as part of the Puzzle class, and if an invalid guess is submitted, throw a Bad\_char exception.

As part of your algorithm in main, catch the Bad\_char exception and respond exactly as in the full credit version – with the error message “Invalid character - try again”.

At this level, the student will also load the game in the gdb or ddd debugger, set a breakpoint, and examine a variable of the student’s choice.

For credit, the Bonus 1 subidrectory will contain the following source code files corresponding to the class diagram: abc1234\_main.cpp, abc1234\_puzzle.h, abc1234\_puzzle.cpp, abc1234\_fuse.h, abc1234\_fuse.cpp. The student will also deliver in this subdirectory a correct Makefile, a abc1234\_puzzle.png image file demonstrating the game in progress, **a** abc1234\_**puzzle-ddd.png image demonstrating the game executing in the ddd or gdb debugger and paused at a breakpoint with at least one variable value visible.**

# Bonus 2 Requirements (10pts)

A game with only one puzzle doesn’t have much replay value.There are 2 lists of 50 words in the text file on blackboard “wordlist.txt” and “wordlist2.txt”. Each word is on a new line.

Add a new class, Words, to add the spice of variety to the game. The constructor Words will accept a filename, and will open that file and load all of the words (one per line) into the private vector of strings named \_words. We’ll need to actually open the text file as a new stream in order to load it into the \_words vector. The Words class contains a single method, get\_word, that returns one of these words as a string at random.

Now modify main to accept as an optional parameter the filename of one of the above list of words or phrases (including possibly spaces). If the parameter is provided, use the Words class to load it into memory and to select one of the words or phrases at random to use as the solution parameter to the Puzzle constructor.

For credit, the Bonus 2 subidrectory will contain the following source code files corresponding to the class diagram: abc1234\_main.cpp, abc1234\_puzzle.h, abc1234\_puzzle.cpp, abc1234\_fuse.h, abc1234\_fuse.cpp, abc1234\_words.h, and abc1234\_words.cpp. The student will also deliver in this subdirectory a correct Makefile, **an abc1234**\_**puzzle-uml.png with the updated class diagram,** and 3 image files – abc1234\_puzzle-1.png, abc1234\_puzzle-2.png, and abc1234\_puzzle-3.png - demonstrating three separate games with different solutions from at least 2 different word list files in progress.