My mastermind

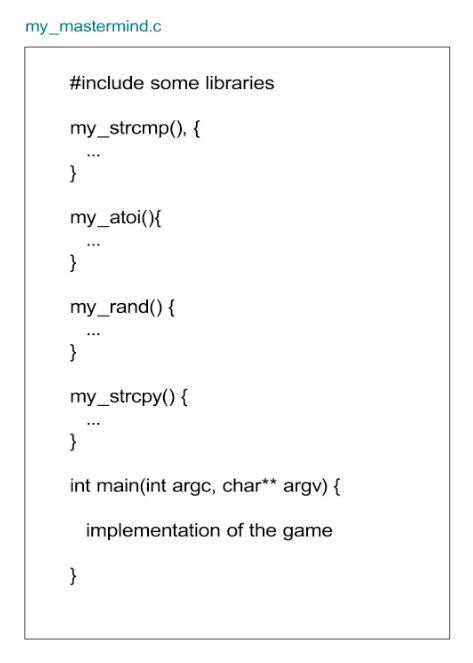
**Game Rules**

Before starting, we need to know what we should implement, so we need to learn about the rules of the game. Here, you can watch a video with an explanation. I think it should be clear enough. <https://www.youtube.com/watch?v=5jtcsBERDEQ>

In our game, we use digits from 0 to 7 instead of colors, and we use such expressions as 'well-placed pieces' and 'misplaced pieces' instead of the red and white pegs shown in the video. For instance, if the secret number is 1234, but the player entered 1243, there will be two well-placed pieces (digits '1' and '2' on their positions) and two misplaced pieces ('3' and '4' exist in the secret number, but they are placed incorrectly). However, if the player enters 1235, there will be three well-placed pieces, and nothing will be said about the piece which is not in the secret number (digit '5').

**Using multiple files with code and Makefile**

Why do we need .h files (header files)?

We could write the whole code in a single file, and it would work fine. However, we can place some functions of code in a separate .c file. For example, when I was doing the task, I placed such functions as my\_strcmp(), my\_atoi(), my\_rand(), my\_strcpy() into a second .c file, just for convenience, because they are not directly connected with the implementation of the game. I could follow the following structure (as shown on the first diagram), but I decided that all these functions are distracting, so I created a new file named additional.c and put them into this file like shown in the second diagram. We need that my\_mastermind.c knows that some functions used there are in additional.c. For that, we create a .h file, which plays a role of a bridge. There are some rules:

1. In .c files, we should include the .h file, similar to how we include libraries, but in double quotes. In this example, it includes "additional.h".
2. .h file should have the following structure:

#ifndef HFILENAME\_H

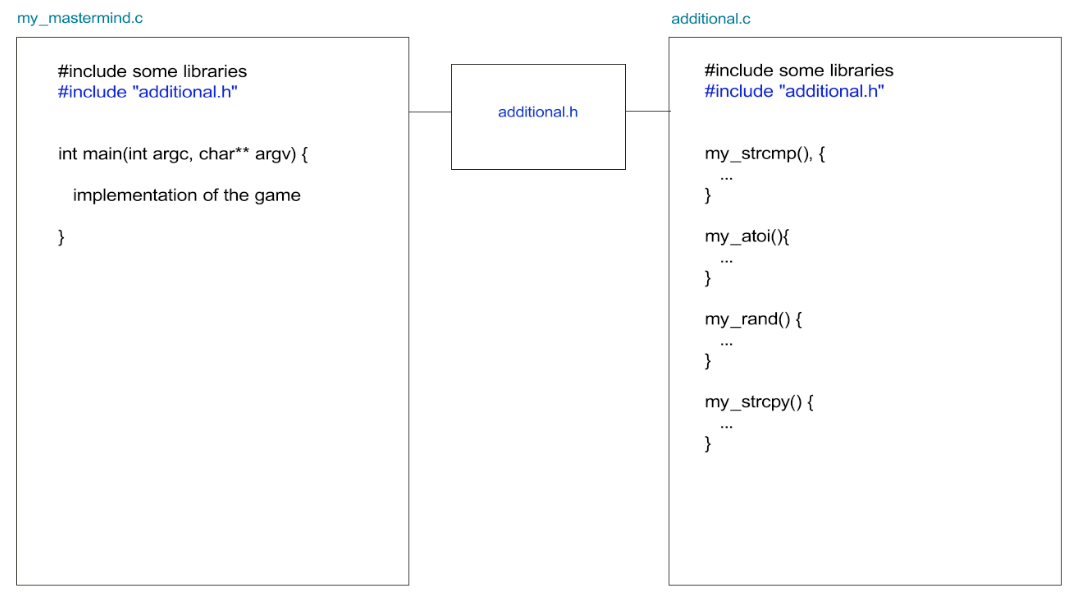
#define HFILENAME\_H

Declarations of functions without implementation

#endif

*Some notes:*

* *After the #infndef and #define keywords, we write the name of the .h file but in capital letters and use underscores (\_) instead of dots or spaces.*
* *The function declaration is a function name with parameters and return type, but without a body. The implementation body is in the .c file (see example here:* [*https://stackoverflow.com/questions/7109964/creating-your-own-header-file-in-c*](https://stackoverflow.com/questions/7109964/creating-your-own-header-file-in-c)*)*

Related links:

<https://stackoverflow.com/questions/7109964/creating-your-own-header-file-in-c>

<https://ravesli.com/urok-21-zagolovochnye-fajly/>

<https://stackoverflow.com/questions/1653958/why-are-ifndef-and-define-used-in-c-header-files>

<https://gcc.gnu.org/onlinedocs/cpp/Once-Only-Headers.html>

<https://www.youtube.com/watch?v=t5JtyDlESns>

Since our program consists of not a single file, we need to compile it properly. So, here comes the Makefile (a link with the explanation is provided in Qwasar in additional resources, but if you need it, I’ll also include it here <https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_make.html> ). You can also watch this video about makefiles: <https://www.youtube.com/watch?v=PiFUuQqW-v8> .

* Having a Makefile, we don’t need to write a command like “gcc -o my\_app main.c foo.c” with all flags and file names to compile the program, but simply write “make” in the terminal, and it will compile.
* Also, we can create a makefile for a single-file code, but, in such a case, for me, it’s more convenient to write a compilation command in the terminal.

Related links.

We can also compile the code not using a makefile, but in the command line, like shown here (however, in this task, we are told to create a makefile): <https://www.linuxtopia.org/online_books/an_introduction_to_gcc/gccintro_11.html>

-Wall -Wextra -Werror and other flags explanation: <https://habr.com/ru/post/490850/>

**Game implementation (how I solved it).**

**General structure**

1. Passing arguments.

Check if the code and number of attempts are written by the user

* 1. If the code is entered manually, store it. Otherwise, randomize and store.
  2. If the number of attempts is entered manually, store the number of attempts. Otherwise, the number will be 10.

1. Game
   1. The user inputs the number to guess it, and the program checks if the input is correct
   2. Compare the number that the user entered with the secret code. Write the number of well-placed and misplaced digits.

Repeat 2 a. and b. the number of attempts times (for-loop). If the player guessed the number, congratulate the player and break the loop.

**A bit more detailed explanation**

1. **Passing arguments.**

The input in the terminal can be different – there can be or can’t be ‘-t’ and ‘-c’ arguments:

* ./my\_mastermind -c [code] -t [attempts]
* ./my\_mastermind -t [attempts] -c [code]
* ./my\_mastermind -c [code]
* ./my\_mastermind -t [attempts]
* ./my\_mastermind

There can also be wrong input, for instance, “./my\_mastermind -c”, but the secret code is not written. If you want to write code that checks such cases and alerts the user that input is wrong, but I suggest supposing that the user passes the arguments correctly in one of the forms above. There is no such requirement for the task, so don’t waste your time.

How to check if the user entered the code and the number of attempts?

The string from the terminal is stored in the array of strings argv[], where each element is each word entered in the command line (separated by spaces). For instance, in input ./my\_mastermind -c 1234 -t 15, argv[0] is “./my\_mastermind”, argv[1] is “-c” and so on. Note: argc is the number of arguments passed from the terminal.

To check if the code is entered, you should find the element “-c” among the elements of argv[]. Just go through the elements of the array argv[] using *for loop* (for (int i = 0; i < argc; i++))and compare each element with “-c”, using a function for comparison of strings (strcmp() in <string.h>, but you should write your own).

If “-c” is found among the elements of argv[], then the next element is the code that we need to guess. Remember that the code in argv[] is stored as a string (array of characters followed by ‘\0’ character), not an integer (it is even advantageous because we can extract each character later to compare it with the code provided by the player). You can store this code in an array of characters by using a function for copying strings (also, it’s better to write own, instead of using <string.h> library). Or you can store the position of the element among argv[].

Do the same with “-t”, but here you will need to convert the number of attempts to an integer using the atoi() function so you can use it in the for loop later.

More about argc and argv in int main(int argc, char\* argv[]) (in Russian): <https://www.andreyolegovich.ru/code/c/argc_argv.php>

If the “-c” argument is found, you either store the secret code’s position among argv[] or save the code in another string. If there is no “-c” argument, the secret code should be randomized, where each of four digits should be generated separately. For randomizing, rand(), srand(), time() functions will be used. As you will generate digits as integers, you will probably need to convert them to characters (use ASCII), depending on whether you store the code as an array of integers or chars.

Here you can read about randomizing: <https://stackoverflow.com/questions/1202687/how-do-i-get-a-specific-range-of-numbers-from-rand>

<https://www.includehelp.com/c-programs/generate-random-numbers-within-a-range.aspx>

When I was doing the task, I forgot to use use srand() and time(), so my randomizer generated the same number over and over again. You can read about the problem more here: <https://stackoverflow.com/questions/1108780/why-do-i-always-get-the-same-sequence-of-random-numbers-with-rand>

In my solution, I checked if the four generated digits of the code are non-repeating, so I used a *while loop* to keep generating if the same digit already exists.

1. **Game implementation**

Create a for loop that repeats the number of attempts times if the “-t” was found. Otherwise, repeat the loop ten times. The rest of the code will be inside the loop.

Print the number of the round. Then prompt the player to enter a number into the command line. Honestly, I cheated in this part and used the scanf() function, but there is no such criterion in peer review, so my solution passed successfully. Accept the entered code as a string (array of chars), then check if each character lies between the 48th and 55th element in the ASCII table (‘0’ and ‘7’ characters respectively). If any of the characters is not in this range or the length of the code is not 4, print a message that the input is wrong and prompt the player to enter the code until it is correct (hint: use do-while loop). I also checked if all digits entered are non-repeating.

As I used scanf(“%s”, enteredNum), the code was immediately stored in the enteredNum string. After the player correctly entered the number, we need to compare the entered number digit by digit with the secret code stored in the beginning. We should compare each character in one array with all characters in another one. If two characters have the same value and the same position in the string, then increment the number of well-placed pieces; if the value is the same, but positions are different, then increment the number of misplaced pieces. If the entered number and the secret code are the same (4 well-placed pieces), then congratulate the player break the loop. Also, consider revealing the code after the last attempt.

That’s it. Good luck in working on the solution! In the end, I’d like to share the contents of my README file. I find it useful.

**my\_mastermind**

***Input format***

To start the game, the user enters a line in terminal in the following format:

./my\_mastermind -c [code] -t [attempts]

* The game will also work if the number of attempts is written first and then goes the code

./my\_mastermind -t [attempts] -c [code]

[attempts] and [code] should be replaced with numbers

* The game will also work if the code and / or the number of attempts are not included in the command

./my\_mastermind -c [code]

./my\_mastermind -t [attempts]

./my\_mastermind

In this case, the code will be a random number and 10 attempts will be given

**Code requirements**

1. It should be a 4-digit number
2. It should not contain repeating elements
3. The elements should be digits from 0 to 7

***Input processing***

**Identifying if the attempts and code entered**

First, we check if the user entered the 4-digit code and the number of attempts by checking

for (int i = 0; i < argc; i++) {

if (my\_strcmp("-c", argv[i]) == 0) {

c = i + 1;

}

if (my\_strcmp("-t", argv[i]) == 0) {

t = i + 1;

attempts = my\_atoi(argv[t]);

}

}

In the code above, we check if any of the arguments contain "-c" or / and "-t" to identify if the user entered the code and the number of attempts. If yes, we store their positions in **c** and **t** variables. Otherwise, these variables will be 0. If there is no "-t" among the arguments, the number of attempts (**attempts** variable) will be kept as 10.

**Code: randomizing or copying from arguments**

The code will be stored in an array of 4 char elements called **num**:

char num[4];

If the variable **c** is equal to zero, which means that the user didn't enter the code, it will be randomized digit by digit:

if (c == 0) {

srand(time(NULL));

int i = 0;

char randDigit;

while (i < 4) {

randDigit = (rand() % 8) + 48;

int same = 0;

for (int j = 0; j < i; j++){

if (num[j] == randDigit){ // elements should be unique

same = 1;

break;

}

}

if (same == 1)

continue;

else {

num[i] = randDigit;

i++;

}

}

}

the code above generates a code with unique elements.

Otherwise, the code will be copied from **argv[c]** to **num**:

else {

for (int i = 0; i < 4; i++){

if (argv[c][i] != 34) {

num[i] = argv[c][i];

}

}

}

***Game***

**The game will repeat in the external *for* loop attempts times:**

for (int i = 0; i < attempts; i++)

**Player input: correct or wrong**

After the game received the code from the player, it cheks if the player entered the code correctly:

1. It includes digits from 0 to 7:

if ((enteredNum[k] < 48) || (enteredNum[k] > 55)){

correctInp = 0;

break;

}

1. It has no repeating elements:

for (int j = 0; j < k; j++){

if (enteredNum[k] == enteredNum[j]){

correctInp = 0;

break;

}

}

**correctInp** stores 0 if the input is wrong and 1 if correct. If it is 0, the program notifies the user that the input is incorrect and prompts the user to enter it until it is correct (**do while** (correctInp != 1) loop).

**Checking if the player guessed the number**

* We compare the secret number with the entered number digit by digit. If there are differences, **correct** variable is assigned to 0. Initially, it was 1.

for (int k = 0; k < 4; k++){

if (enteredNum[k] != num[k]) {

correct = 0;

break;

}

}

* If the **correct** variable is 1, it means that the user guessed the number, so we congratulate him/her and break the external loop, and the game finishes.

if (correct == 1) {

printf("Congratz! You did it!\n");

break;

}

* In the final nested loop, we count the number of misplaced and well-placed pieces according to the positions of the digits in the **enteredNum** and **num** arrays.

for (int eni = 0; eni < 4; eni++){

for (int ci = 0; ci < 4; ci++){

if ((ci == eni) && (num[ci] == enteredNum[eni])){

wpp++;

}

else if (num[ci] == enteredNum[eni]) {

mpp++;

}

}

}

**correct** variable stores 0 if the player didn't guess and 1 otherwise.

**mpp** and **wpp** variables stand for misplaced and well placed pieces.