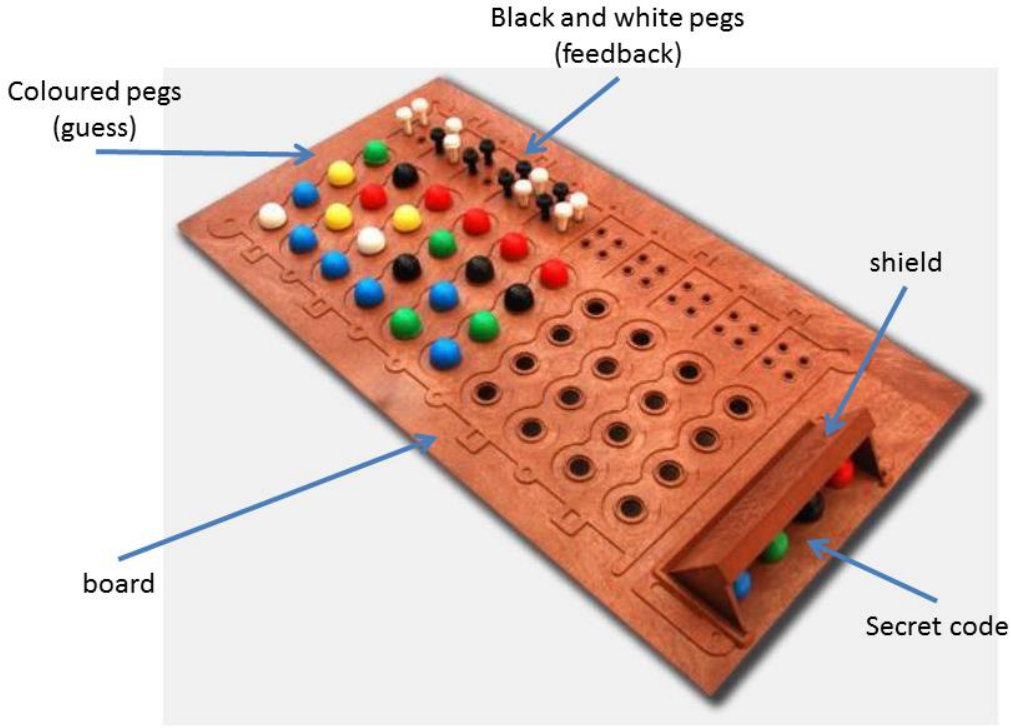
Final Project: *MasterMind World Championship*

Introduction

MasterMind is a code breaking game for two players, played on a special board (see image). One player becomes the *codemaker*, and other the *codebreaker*. The *codemaker* chooses a pattern of four coloured pegs (the secret code), hiding it from the *codebreaker* using a shield. The codebreaker tries to guess the secret code in several turns, with the codemaker providing feedback after each turn.

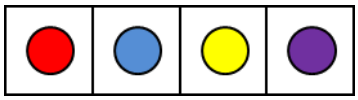


Each guess is made by placing a row of code pegs on the decoding board. The feedback is provided by placing black and white key pegs in the small holes of the row with the guess, according to the following rules:

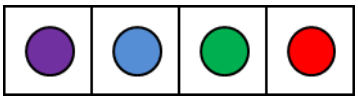
- A black key peg is placed for each code peg from the guess which is correct in both colour and position.
- A white key peg indicates the existence of a correct colour code peg placed in the wrong position

For instance, if the secret code chosen by the codemaker is the pattern to your left and the guess made by the code breaker is the guess to your right, the feedback provided will be one black peg (because the blue peg is in the right position) and two white pegs (because purple and red pegs are part of the code, but they are in the wrong position)

Secret code



Guess



Once feedback is provided, another guess is made; guesses and feedback continue to alternate until either the codebreaker guesses correctly, or ten incorrect guesses are made.

Objective

The objective of the final project is to develop a program to run a MasterMind championship, where games are played the computer.

The computer will play the role of the codemaker and the user will play the role of the code breaker.

The project is divided in **four parts**. Each part extends what was developed before incorporating new functionality (each part would be called a "sprint" in software development). These parts are play game; manage games; manage players; and final integration.

Part 1. Play game

The colours of the pegs will be represented as integer numbers from 1 to 6 (6 possible colours). To make things easy for the codebreaker, **duplicate colours** in the secret code are **not allowed** (each digit can only turn up once in the secret code)

After each Master Mind match, a score is awarded to the codebreaker as follows:

- 100 points if the code is broken in the first attempt.
- For each additional attempt, the score will be reduced by 10 points, therefore if the user guesses the secret code in the first attempt the score will be 100 and if he guesses it in the 10th attempt the score will be 10.
- If the player is not able to guess the code in 10 attempts, the score will be 0 points.

Example game

Secret Code: **1 2 6 4**

*	*	*	*	--	bk	wh
5	1	3	2	--	0	2
1	3	4	6	--	1	2
1	5	3	4	--	2	0
1	6	2	4	--	2	2
2	1	6	4	--	2	2
1	2	6	4	--	4	0
-	-	-	-	--	-	-
-	-	-	-	--	-	-
-	-	-	-	--	-	-
-	-	-	-	--	-	-

Congratulations! Your score is 50

Template

Download from Aula Global the template to build your project (*mastermind.c*). It is a C file with the basic structure of Part 1 of the project. It includes the headers of the functions that you have to develop. **You should not change these headers.**

Step 1. verifyCode

Develop the function `verifyCode` to compare the guess to the secret code, and return the feedback (number of black and white pegs).

The prototype of this function is:

```
int verifyCode(int secretCode[], int guess[], int *blacks, int *whites)
```

where `secretCode` is the secret code (1x4 vector), `guess` is the vector containing the guessed code (1 x 4 vector) and white and feedback is the feedback (scalars).

Test this version with different values for `secretCode` and `guess`. To do so, use the function `generateSecretCode` provided in the template, that generates a random secret code).

Step 2. Play the game

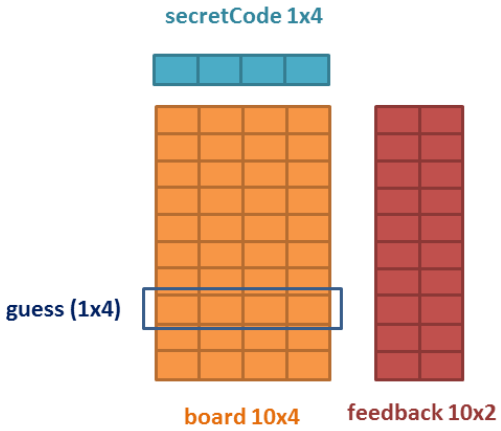
Implement the game, asking the user for a guess (four digits) and providing feedback until she guesses correctly, or makes ten incorrect guesses.
To verify the code, use the function `verifyCode` developed in Step 1. To generate the code use `generateSecretCode`.
The successive guesses will be stored at different rows of a 10 x 4 matrix `board`. The feedback will be stored in a similar matrix (10 x 2) called `feedback`. After each attempt, display the current state of the board (all the board including guesses and feedback. Use the letter '_' to represent the positions of the board that have not been used so far).
When the user correctly guesses the secret code, the program must display the score, the message "congratulations" and the final state of the board.

Develop a function to display the board, with this header (that is included in the template)

```
void displayBoard (int board[ATTEMPTS][SIZE], int feedback[ATTEMPTS][2],
                   int nRows)
```

The function gets as input parameters a 10 x 4 matrix board, a matrix (10 x 2) feedback and a number representing the number of attempts (corresponding to the number of lines that must be displayed)

Summary: variables to be used



Step 3. Define game structure

We are now going to group all the information related to the game in a structure with the following definition and initialization

```
struct typeGame {
	int nAttempts;
	int secretCode[4];
	int board [10][4];
	int feedback [10][4];
	int score;
};
```

Modify your code to adapt to this change. The main difference will be to use `game.secretCode` instead of `secretCode`; `game.board` instead of `board`, and so on.

It is important that you use exactly these names for the fields of the structure, to make your code compatible with the initialization functions we will provide

Step 4. Function play

Group all the instructions related to playing the game in a function with the following definition and initialization

```
struct typeGame play(struct typeGame g);
```

where the input and output are the same structure of type game. The input structure is contains the secret code, and the same structure is returned, but now it contains all the information after playing the game (`board`, `feedback`, `score` and `nAttempts`)

Step 5. Function displayGame

Build a function to display the game, Including score, number of attempts, secret code board and feedback. Reuse the code you already have to display the game and organize it in a function with the following header.

```
void displayGame (struct typeGame g);
```

The function must have as input parameter a structure of type game.

You can call the function displayBoard to display the board.

Part 2. Generalizing to a vector of games

The objective of this sprint is to add functionality to store information regarding different games played in the championship. This information will be stored in a vector of structures of type `typeGame`.

Step 1. Declare a the vector of structures games

The vector will store a maximum of 50 games. This value will be declared as a constant MAX_GAMES.

```
struct typeGame games[MAX_GAMES];
```

Step 2. Menu

When the program starts, it will now display a menu with three options as shown below. The user will select one option, the selected option will be run, and then the menu will be displayed again. This will continue until option Exit is selected. For the moment, just add the loop and display the option selected, and exit when the user selects 0.

```
1. Display all games
2. Play game
0. Exit
```

Step 3. Integrate play game with the structure of games

When the user selects option 2, the program must call function `play` (from Part 2) to play the game. Before this call to `play`, the secret code must be initialized (using `generateSecretCode` function from Part 1). The output of `play` is a structure game, that must be stored in the first empty position in the vector of games. To assign the game to the right position of the vector, you need a counter (`nGames`) to keep track of the number of games registered so far. Before playing a game, you need to check that there is room in the vector for one more game.

Step 4. Display list of games

When the user selects option 1, all the games registered in vector games will be displayed in the form of a table. Only the summary information of each game is shown now (secret code, number of guesses and score but not the board or the feedback). In the next part we will also show what player played the game (but forget this for now).

Implement the following function to display the games, where `games` is the vector of games, and `n` the number of games registered

```
void displayListOfGames (struct typeGame listG [], int nGames);
```

Part 3. Initialization of players and ranking

Introduction

In this part we will add information on the players playing the games and add code to display the ranking information of the championship

Step 1. Include the library player.c

You will use several functions to generate initialization data and to display player information. These functions are already implemented, and you only need to add them to your project, following these steps:

1. Download two files from Aula Global: *players.c* and the corresponding headers *players.h*. Copy them to the folder where your program is.
2. Add these two files to your DevC++ project (for example in the right window, add to project)
3. Write the code to include the new functions in the library players. This means adding an include directive similar to the ones we have been using to include other libraries such as stdio. The line you need to add is:

   ```
   #include "players.h"
   ```

The library player contains the following:

- declaration of the constants needed in the program
- Three functions to generate data and display players:
  ```
  void displayListOfPlayers(struct typePlayer listP[],int nPlayers);
  void loadListOfGames(struct typeGame listG[], int *nGames);
  void loadListOfPlayers(struct typePlayer listP[],int *nPlayers);
  ```
- declaration of the structure players:
  ```
  struct typePlayer{
         int id;                 // player id
         char name[256];
         char surname[256];
         int score;              // score
         int nGPlayed;           // number of games played
         };
  ```
- declaration of the structure games.

Since the structure `typeGame` is now declared in the headers file, you have to remove it from your main.c. To be able to use the initialization functions, the name of the structure `typeGame`, as well as the names of its fields in your main.c, must be the same as the ones used in the headers file.

Step 2. Declare the vector of players and load initialization data using the functions provided

We will add now another vector of structures to store information about the players. The structure is described in the previous step.

Calling the functions `loadListOfPlayers` and `loadListOfGames` available `in players.c`, load initial data into your vectors of games and of players. This must the first action performed in your program, before displaying the menu. Consider that both functions have a parameter that represents the number of players or games actually loaded.

Test that you are loading data properly using your function `displayListOfGames` and the function `displayListOfPlayers` available in players.c

Step 3. Modify game to include information on the player

The program needs to register which player is playing each game. To do so, extend the structure typeGame to add a new field called playerId. Player ids start in 1 (not 0). In the next step you will see how the playerId is used.

Step 4. Integrating players with games: update your code to register which player plays one game

Extend your program to store id of the player playing each game. That is, before playing a game, get the player id (an integer) from the user. Check that the player id is valid (from 1 to current number of players), and update this information in the game structure before passing it to function playGame.

Step 5. Add new options to the menu

Add two new options to the program menu

```
3.  Display list of players
4.  Display ranking of players
5.  Display top players
```

Step 6. Update the player's scores

To be able to display the top players of the championship, we need to know their scores. We will do this with a function updatePlayersScores that will take information from the vector of players and update the vector of games. Input parameters are be the vector of games, the vector of players, the number of games and the number of players

```
void updatePlayersScores (struct typeGame listG[], struct typePlayer listP[],
int nGames,int nPlayers){
```

The function must first initialize the score and number of games played by every player to zero, and then use the information in the vector of games to update the score and number of games of each player. Consider that the position of a player in the vector of players is related to the player id: player in position 0 will have id 1, player in position 1, id 2 and so on. This means that, for a game played by player 6, you must add the corresponding score to the player in position 5.

Note that there are other more efficient ways to implement this, such as updating the score every time a new game is played. We will not do this to simplify the project, instead, we will recalculate all the scores before displaying the players, every time we want to see the players.

Step 7. Sort players and display ranking

Implement a function to sort the players according to their score. Use the bubble algorithm for sorting.
Before sorting, copy all the players to a new structure (`sortedPlayers`) and sort this copy. Do not sort the original vector, because you need to keep the original order for other functions (such as `updatePlayersScores`).

Next, display the ranking of players: all players sorted by their score. To do this, first update the scores (function `updatePlayersScores)`, next copy the data to a new vector, sort this vector by score, and finally display the sorted list (use function `displayListOfPlayers)`.

Step 8. Display top players

Add code to display the top players. The user will select how many players to display (for example, top 3 players). You must use the `updatePlayersScores, sortPlayers` and `playListOfPlayers` functions to do this.

Final remarks

Do not change the header of any of the functions unless you are explicitly asked to do so. This is important to help us grade the project, and is in line with software development practice (when you work with a library of functions or in a team of developers, you will not be able to modify the headers)

Instructions for turning in the final assignment

You can re-submit your project any number of times before the deadline. **We recommend you upload a version in advance and then update it if you make last minute changes**.

You can only upload a file per group. The program must start with comment lines, including the group number and the student names and NIAs, with this format:

```
/*
Group 12
Ana Gómez Pérez 100023456
Juan Rodríguez López 100098765
John Smith 100098765
*/
```

If the project is not working properly, or if some part of it is not implemented or some part doesn't work, you must add some comments with details on this. Describe what is not implemented / not working. This will help us grading the project.