# MP5 Codebreaker

Due Wednesday, October 1, 2014 at 10:00 p.m.

## Overview

Your task this week is to implement the logic for a code-breaking game. A code sequence of four numbers from 1 to 8 is first chosen at random. The player guesses a sequence of four numbers and is given feedback on each guess. This feedback includes the number of values that appear in the same place in the solution code (these we call perfect matches), the number of values less than their corresponding number in the solution code , and the number of values greater than the their corresponding number in the solution code. If the player manages to guess the correct sequence in ten or fewer guesses, they win the game.  Otherwise, they lose.

The objective for this week is for you to gain some experience with basic I/O, to implement code using multiple subroutines, and to solve a problem that requires moderately sophisticated reasoning and control logic.

The task here is thus somewhat more difficult than last week's. If you want to read ahead and use arrays, feel free to do so. But the code should be fairly manageable without using arrays (we wrote the solution that way, just to check).

## The Pieces

The four files that you should examine are the codebreaker header file, the codebreaker source file that you must complete, the main file, and the make file.

Let's discuss each of these in a little more detail:

- main.c - This contains the program's main function. We give you this code. The main function will call the functions you write in codebreaker.c to play the game.
- codebreaker.h - This header file provides function declaration and descriptions of the functions that you must write for this assignment.
- codebreaker.c - The source code file for the functions necessary to play the game. Function headers for all the functions are provided to help you get started. We will work together on the `set_seed` function in programming studio. The next two, `start_game` and `make_guess` are left for you to finish independently.
- Makefile - this includes the make configuration to build the mp5 program.

We have also included some additional files for testing, but we leave discussion of those files for the Testing section of this document.

In programming studio, we will discuss pseudo-random number generation and the idea of error-checking user input, then develop the `set_seed` function together to robustly translate user input into a well-defined seed for the pseudo-random number generator.

**The rest of the assignment must be completed on your own.** If you want to get started before programming studio, you can leave the implementation of `set_seed` provided in the distribution and work on `start_game` and `make_guess`. Together with `set_seed`, these three functions comprise the whole assignment for this week.

# Details

You should read the descriptions of the functions in the header file and peruse the function headers in the source file before you begin coding.

For the `set_seed` routine that we will develop in programming studio, the function signature appears below:

```
int set_seed (const char* seed_str);
```

The function receives a string (a pointer to a character) as its input and produces a 32-bit signed integer as output. The "`const`" qualifier means that the routine is not allowed to change the contents of the string.

We use the following two library calls to produce sequences of pseudo-random numbers:

```
void srand (unsigned int seed);
int rand();
```

The user enters the value in the console and it's delivered to the `set_seed` function as a string when the player presses <Enter>. The `set_seed` routine must then check whether the string represents a number, and, if so, use it to seed the pseudo-random number generator via a call to `srand`. Calls to `rand` then produce pseudo-random numbers in the range $[0, 2^{31} - 1]$.

The return value from `set_seed` indicates whether the input string did in fact correspond to a number. When the string represents a number, the function returns 1. Otherwise, the function returns 0.

We use fixed seeds in this assignment because a fixed seed produces a fixed sequence of random numbers, which means that the solution code will always be the same. Determinism makes debugging much easier - imagine trying to find a bug that only appears in one run out of every million.

Now let's consider the part that you must do alone. Here are function signatures for the two functions:

```
int start_game (int* one, int* two, int* three, int* four);
int make_guess (const char* guess_str, int* one, int* two, int* three,
                int* four);
```

The `start_game` routine selects the solution code, a set of four numbers from 1 to 8. To ensure consistency between your program's output and ours, you **must use the following algorithm for generating the solution code**.

**Step 1:** Starting with the first value in the code sequence, generate a random integer in the range 0 to 7 inclusive using a single call to `rand` and a single modulus (%) operator.
**Step 2:** Add 1 to the resulting number to get a value in the range 1 to 8.
**Step 3:** Repeat for the other three solution code values (in order).

You must also make your own copy of the solution code using file-scoped variables. This copy will be necessary when you implement `make_guess`.

Be sure not to call `srand` outside of the `set_seed` function and not to call `rand` outside of the `start_game` function. Calling either of these disrupts the sequence of random numbers and will cause your output to differ from ours.

Finally, you must write the `make_guess` routine that compares a player's guess with the solution. The inputs to this routine include a string (the player's input) and four pointers to integers.

Your routine must validate the string in the same way that we did for `set_seed`. A valid string contains exactly four numbers (and no extra garbage at the end). All four numbers in the string must be between 1 and 8. If the string is invalid, your routine must print an error message, "`set_seed: invalid seed\n`", then return 0.

For a valid string, you must store a copy of the guessed code *in order* in the four addresses provided as input parameters.

Your routine must then compare the guessed code with the solution code to count the number of perfect, high, and low matches, then print a message informing the player of the results.

Let's consider some examples. Imagine that the solution code is 1 1 2 3. If the player guesses 1 2 2 4, the first (leftmost) and third code values match perfectly, so they have two perfect matches. The second and fourth code values are greater than the corresponding number in the solution. So there are two high matches and zero low matches.

What happens if (using the same solution code: 1 1 2 3 the user guesses 4 4 1 1? Here they have no perfect matches. How many high and low matches matches does the guess code have? Once your routine has calculated the number of perfect, high, and low matches, it must print a message using exactly the format shown here (without the quotes):

"With guess 1, you got 0 perfect matches, 2 high, and 2 low.\n"

The guess number starts at 1 and goes up as high as 10 - your code must also track this value using a file-scoped variable. Note that only valid guesses count as turns. Do not adjust the word "matches" for subject-verb agreement. (i.e. leave "1 perfect matches" as is).

The `make_guess` routine should return 1 when the user has provided a valid guess string.

# Specifics

- Your code must be written in C and and must be contained in the `codebreaker.c` file provided to you. We will NOT grade files with any other name.
- You must implement `set_seed`, `start_game`, and `make_guess` correctly.
- Your routine's return values and outputs must match the gold version's exactly for full credit.
- Your code must be well-commented. You may use either C-style (/* can span multiple lines */) or C++-style (// comment to end of line) comments, as you prefer. Follow the commenting style of the code examples provided in class and in the textbook.

# Building and Testing

As with MP4, all operations mentioned here should be performed from your MP5 directory.

You should test your program thoroughly before handing in your solution. We have provided you a set of tests, but you should get in the habit of writing your own tests.

To compile the program, type:
`make`

If successful, the compiler produces an executable called `mp5`, which you can execute by typing:
`./mp5`

You can also run the code with the gdb debugger by typing:
`gdb mp5`

The testFiles subdirectory contains some sample input and output files. These are not scripts, but direct input.  If you want to test your program with the input files type:

./mp5 < testFiles/input1 > myout1

Then diff the myout1 file with test-examples/output1.

# Grading Rubric

*Functionality (65%)*

- 10% - `set_seed` function works correctly
- 10% - `start_game` function works correctly
- 40% - `make_guess` function works correctly
- 5% - all outputs match exactly

*Style (15%)*

- 5% - compilation generates no warnings (note: any warning means 0 points here)
- 5% - does not use global variables (file-scoped are necessary)
- 5% - indentation and variable names are appropriate and reasonably meaningful (index variables can be single-letter)

*Comments, clarity, and write-up (20%)*

- 5% - introductory paragraph explaining what you did (even if it's just the required work)
- 15% - code is clear and well-commented

Note that some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.