Software Technology Department

Simplified Checkers

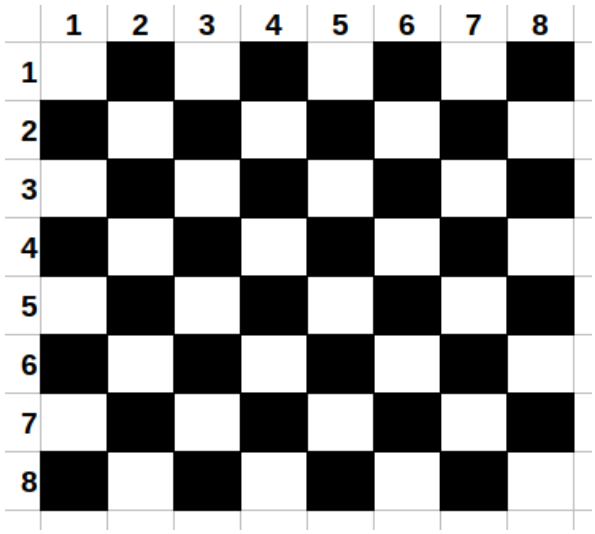
CCPROG1 MACHINE PROJECT (Deadline of submission on/before December 5)

Board games are really entertaining resulting in its huge popularity giving the global board games market over $7.2 billion worth and is even expected to increase to $12 billion by 2023 [1]. One popular type of board game that has a long history is checkers. It was believed that the oldest form of the game checkers was played in an ancient city Ur in Iraq around 3,000 B.C.E. after archeologists unearthed an artifact. There was also another version of the game called Alquerque played in ancient Egypt dating back to 1,400 B.C.E. The standard game of checkers used today was adapted by a Frenchman around the start of the 12th century where it was first played on a chessboard with 12 pieces per player. It was then in 1756 that an English mathematician stated the official rules of the game and became known as 'Checkers'. The Philippines has also adapted this game and called it 'Dama'. And in 1952, the game was first computerized - not only to simulate the game but to actually win.

In an attempt to appreciate the long history of checkers even before it was computerized, a simplified version of the game will be created. Your problem analysis and logic formulation will be practiced in developing this program. The game itself has provided significant training in thought and logic to students.

BOARD

Similar to a chessboard, a checkers board consists of a checkered 8x8 grid. According to the official game rules, only the black squares are used where pieces can be positioned.



[1] https://www.statista.com/statistics/829285/global-board-games-market-value/

BOARD PIECES

There are 24 game pieces, 12 for each player. One color should be dark and the other light.

GAME PLAY

Each player (**Player A** and **Player B**) begins with 12 pieces. Owner of black pieces (meaning the first player) should be randomized. Players then take turns in moving their pieces.

MOVES

Pieces can only move diagonally and are limited to forward moves.
A piece making a non-capturing move may only forward to one square.

CAPTURING

In this simplified version, only one capture is allowed in each move.
A piece can capture the opponent's piece by leaping over it and landing on the available straight diagonal on the other side given that it is empty.

WINNING

In this simplified version, a player wins once:

- ★ his piece reached the end row of the opponent's side, or
- ★ the opponent cannot make a move due to blocked pieces, or
- ★ he captured all the opponent's pieces.

YOUR TASK

Implement the simplified version of Checkers game in C considering the above specifications and the following statements:

1. A welcome page displaying the menu (*Play*, *View scores*, *Quit*) is shown.
2. After choosing *Play*, the screen should display who gets the black pieces - meaning the first to play/move his piece. This should be randomized (as if flipping a coin) by using the *rand()* in C.
3. Initially, the board is displayed with D1-D12 for dark pieces, L1-L12 for light pieces, and any representation for the remaining blank black squares (this can be a black square using ASCII code[2] or any symbol you like). Remember that we only use the black squares of the board so after moving a piece, the blank square should also display a black square representation.
4. During game play, the in-turn player should be asked which piece to move and to where it will be positioned. Your program must be able to check whether this is a possible move considering that the landing position is empty and diagonally adjacent to the current position.
5. A capturing move must also be (automatically) checked if acceptable - meaning the 'captured' opponent piece is diagonally adjacent to the moving piece and this moving piece is landing on an empty black square.

[2] https://www.ascii-code.com/

6. A game piece count showing each player's available pieces on the board should also be displayed which updates after every capture of an opponent's piece.
7. A surrender option should be available to the in-turn player which results in his opponent winning the game and increasing his score (+1).
8. After a game, a winning screen should be displayed congratulating the winning player.
9. 'View Scores' menu should display the scores of the two players - **Player A** and **Player B**.
10. 'Quit' option should display a closing screen and/or message before terminating the whole program.

QUESTIONS TO PONDER:

1. What is/are the input to your program during the game and why did you design it so?
2. How will you display/output your board?
3. How will you move the pieces?
4. How will you handle the surrendering of a player?
5. How will the pieces be monitored - keeping count of those that are still available and are already captured?

How to Approach the Machine Project

Step 1: Problem analysis and algorithm formulation

Read the MP Specifications again! Identify clearly what are the required information from the user, what kind of processes are needed, and what will be the output(s) of your program. Clarify with your professor any issues that you might have regarding the machine project.

When you have all the necessary information, identify the necessary functions that you will need to modularize the project. Identify the required data of these functions and what kind of data they will return to the caller. Write your algorithm for each of these modules/functions as well as the algorithm for your main program.

Step 2: Implementation

In this step, you are to translate your algorithm into proper C statements. While implementing, you are to perform the other phases of program planning and design (discussed in the other steps below) together with this step.

Follow the coding standard indicated in the course notes (Modules section in AnimoSpace).

You may choose to type your program in a text editor or an IDE (i.e. Dev-C IDE) at this point. Note that you are expected to use statements taught in class. You can explore other libraries and functions in C as long as you can clearly explain how these work. You may also use arrays, should these be applicable and you are able to properly justify and explain your implementation using these. For topics not covered, it is left to the student to read ahead, research, and explore by himself.

Note though that you are **NOT ALLOWED** to do the following:

- to declare and use global variables (i.e., variables declared outside any function),
- to use `goto` statements (i.e., to jump from code segments to code segments),
- to use the `break` statement to exit a block other than `switch` blocks,
- to use the `return` statement or `exit` statement to **prematurely** terminate a loop or function or program,
- to use the `exit` statement to **prematurely** terminate a loop or to terminate the function or program, and
- to call the `main()` function to repeat the process instead of using loops.

It is best that you perform your coding “incrementally.” This means:

- dividing the program specification into subproblems, and solving each problem separately according to your algorithm;
- coding the solutions to the subproblems one at a time. Once you’re done coding the solution for one subproblem, apply testing and debugging.

Documentation

While coding, you have to include internal documentation in your programs. You are expected to have the following:

- File comments or Introductory comments
- Function comments
- In-line comments

Introductory comments are found at the very beginning of your program before the preprocessor directives. Follow the format shown below. Note that items in between **< >** should be replaced with the proper information. Items in between ***[ ]*** are optional, indicate if applicable.

```
/*
     Description: <Describe what this program does briefly>
     Programmed by: <your name here>  <section>
     Last modified: <date when last revision was made>
     Version: <version number>
     [Acknowledgements: <list of sites or borrowed libraries and sources>]
*/

<Preprocessor directives>

<function implementation>

int main()
{
     return 0;
}
```

Function comments precede the function header. These are used to describe what the function does and the intentions of each parameter and what is being returned, if any. If applicable, include pre-conditions as well. Pre-conditions refer to the assumed state of the parameters. Follow the format below when writing function comments:

```
/*    <Description of function>
     Precondition: <precondition / assumption>
     @param <name> <purpose>
     @return <description of returned result>
*/
<return type>
<function name> (<parameter list>)
     :
```

Example:

```
/* This function computes for the area of a triangle
   Precondition: base and height are non-negative values
   @param base is the base measurement of the triangle in cm
   @param height is the height measurement of the triangle in cm
   @return the resulting area of the triangle
*/
float
getAreaTri (float base,
            float height)
{
    ...
}
```

In-Line comments are other comments in major parts of the code. These are expected to explain the purpose or algorithm of groups of related code, esp. for long functions.

Step 3: Testing and Debugging

SUBMIT THE LIST OF TEST CASES YOU HAVE USED.

For each feature of your program, you have to fully test it before moving to the next feature. Sample questions that you should ask yourself are:

1) What should be displayed on the screen after a user input?
2) What would happen if inputs are incorrect? (e.g., values not within the range)
3) Is my program displaying the correct output?
4) Is my program following the correct sequence of events (correct program flow)?
5) Is my program terminating (ending/exiting) correctly? Does it exit when I press the command to quit? Does it exit when the program's goal has been met? Is there an infinite loop?
6) and others...

IMPORTANT POINTS TO REMEMBER:

1. You are required to implement the project using the C language (**C99** and NOT C++). Make sure you know how to compile and run in both the IDE (DEV-C++) and the command prompt via

   `gcc -Wall` *<yourMP.c>* `-o` *<yourExe.exe>*

2. The implementation will require you to:
   - Create and Use Functions

     Note: Non-use of self-defined functions will merit a grade of **0** for the **machine project**.
   - Appropriately use conditional statements, loops and other constructs discussed in class (Do not use brute force solution. You are not allowed to use `goto` label statements, `exit` statements. You are required to pass parameters to functions and not allowed to declare global or static variables.)
   - Consistently employ coding conventions
   - Include internal documentation (i.e., comments)
3. Deadline for the project is the **8:00 AM of December 5, 2022 (Monday)** via submission through **AnimoSpace**. After this time, the submission facility is locked and thus no MP will be accepted anymore and this will result to a **0** for your **machine project**.

4. The following are the deliverables:

Checklist:

- Upload in **AnimoSpace** by clicking **Submit Assignment** on Machine Project and adding the following files:
  - source code*
  - test script**
- email the softcopies of everything as attachments to YOUR own email address on or before the deadline

Legend:
* Source Code also includes the internal documentation. The **first few lines of the source code** should have the following declaration (in comment) **BEFORE** the introductory comment:

```
/*****************************************************************
This is to certify that this project is my own work, based on my
personal efforts in studying and applying the concepts learned. I
have constructed the functions and their respective algorithms
and corresponding code by myself. The program was run, tested, and
debugged by my own efforts. I further certify that I have not
copied in part or whole or otherwise plagiarized the work of other
students and/or persons.

<your full name>, DLSU ID# <number>
*****************************************************************/
```

** Test Script should be in a table format, with header as shown below. There should be at least 3 distinct test classes (as indicated in the description) **per function**. There is no need to test functions which are only for screen design.

Function Name	**#**	**Test Description**	**Sample Input** *(either from the user or passed to the function)*	**Expected Result**	**Actual Result**	**P/F**
`getAreaTri()`	1	base and height measurements are less than 1	base = 0.25 height = 0.75	...	...	...
	2					
	3					

5. **MP Demo:** You will demonstrate your project on a specified schedule during the last weeks of classes. Being unable to show up on time during the demo or being unable to answer convincingly the questions during the demo will merit a grade of **0** for your **machine project**. The project is initially evaluated via black box testing (i.e., based on output of running program). Thus, if the program does not compile successfully using gcc and execute in the command prompt, a grade of 0 for the project will be incurred. However, a fully working project does not ensure a perfect grade, as the implementation (i.e., correctness and compliance in code) is still checked.
6. Any requirement not fully implemented and instruction not followed will merit deductions.
7. This is an **individual project**. Working in collaboration, asking other people's help, borrowing or copying other people's work or from books or online sources (either in full or in part) are considered cheating. Cheating is punishable by a grade of **0.0** for **CCPROG1** course. Aside from which, a cheating case may be filed with the Discipline Office.
8. **Bonus points:** You can earn up to **at most 10 points** bonus for additional features that your program may have. Sample features and implementation that may allow you to gain bonus points include: **allowing multiple captures of the opponent's game pieces**. Some of the indicated additional features may require self-study. Also, any additional feature not stated here may be added but **should not conflict with whatever instruction was given in the project specifications**. Bonus points are given upon the discretion of the teacher, based on the difficulty and applicability of the feature to the program. Note that **bonus points can only be credited if all the basic requirements are fully met** (i.e., complete and no bugs).

HONESTY POLICY AND INTELLECTUAL PROPERTY RIGHTS

Honesty policy applies. Please take note that you are NOT allowed to borrow and/or copy-and-paste – in full or in part any existing related program code from the internet or other sources (such as printed materials like books, or source codes by other people that are not online). **You should develop your own codes from scratch by yourself.**