ECE 209
Fall 2010

Program 2: `yzscore`

See handout for due date

Due Wednesday, Oct 6, 2010 @ 11:45pm

This programming assignment must be completed individually. Do not share your code with or receive code from any other student. Evidence of copying or other unauthorized collaboration will be investigated as a potential academic integrity violation. **The minimum penalty for cheating on a programming assignment is a grade of -100 on the assignment.** If you are tempted to copy because you're running late, or don't know what you're doing, you will be better off missing the assignment and taking a zero. Providing your code to someone is cheating, just as much as copying someone else's work.

DO NOT copy code from the Internet, or use programs found online or in textbooks as a "starting point" for your code. Your job is to design and write this program from scratch, on your own. Evidence of using external code from any source will be investigated as a potential academic integrity violation.

This program is based on a game called Yahtzee™. In the game, the player rolls five dice, and a score is assigned based on the values and patterns of the dice. We're not going to implement the entire game, but this program will calculate various scores for a particular roll.

The learning objectives of the program are:

- Use an array to track multiple values of the same type.
- Split a program across multiple files.

Background

Yahtzee is a dice-rolling game. Each player rolls five dice and calculates the best score for that roll. The player then has the option to re-roll some or all of the dice to improve the score. At the end of the turn, the player records the score for the roll in one of several categories, and the game proceeds until all of the categories are filled, and then a final score is computed.

For this program, we will only consider a single roll of dice. (No re-roll.) We will calculate and report the score for that roll for each scoring category. (For those who know Yahtzee, we will only consider categories from the bottom half of the score card.)

The scoring categories are shown in Figure 1. If the rolled dice match the pattern in the category, the score is calculated as shown. If the dice do not match, then the score for that category is 0.

Note that a particular roll may match more than one category. For example a roll of 3, 2, 3, 2, 3 can be scored as three-of-a-kind or as a full house. (Any roll can be scored as Chance.) Our program will take a single roll and compute (and print) the scores for all seven categories.

This program will require you to use one or more arrays of integers. We will be discussing arrays and pointers in class, but you can also read Chapter 16 of P&P and Chapter 10 of CPP.

Chapter 10 of C Primer Plus will be sufficient

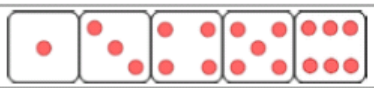
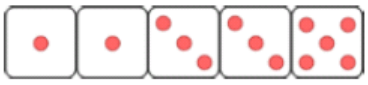
Category	Description	Score	Example
Three-Of-A-Kind	At least three dice showing the same face	Sum of all dice	
Four-Of-A-Kind	At least four dice showing the same face	Sum of all dice	
Full House	A three-of-a-kind and a pair	25	
Small Straight	Four sequential dice (1-2-3-4, 2-3-4-5, or 3-4-5-6)	30	
Large Straight	Five sequential dice (1-2-3-4-5 or 2-3-4-5-6)	40	
Yahtzee	All five dice showing the same face	50 First Yahtzee only	
Chance	Any combination often acts as discard box for a turn that will not fit in another category (thus the name), although during a lucky game it can be used to record a high score	Sum of all dice	

Figure 1. Yahtzee scoring categories. (http://en.wikipedia.org/Yahtzee)

Program Specification

You will be given a C file named **main.c**, containing the main function of the program. And you will be given a header file, **yzscore.h**, that contains declarations for all of the functions that you will implement. The main function will handle all of the user interface issues (printing results, getting values from the user). Your job is to create a C file named **yzscore.c**, which will contain the implementations (definitions) of the functions.

There are seven functions for you to write. The header file **yzscore.h** contains the following declarations:

```
int score3(int dice[]); /* score of dice for 3-of-a-kind pattern */
int score4(int dice[]); /* score of dice for 4-of-a-kind pattern */
int score5(int dice[]); /* score of dice for Yahtzee pattern */
int scoreFull(int dice[]);  /* score of dice for full house pattern */
int scoreSmall(int dice[]); /* score of dice for small straight pattern */
int scoreLarge(int dice[]); /* score of dice for large straight pattern */
int scoreChance(int dice[]); /* score of dice for chance pattern */
```

In each case, the caller passes in a five-element integer array, containing the dice values (between 1 and 6) from the user. There is a function for each scoring category. If the dice match the pattern, then the score is computed according to Figure 1. If the dice do not match the pattern for that category, the function returns 0.

Example: An array containing the values 4, 4, 1, 4, 4 is passed to the **`score3`** function. The dice do match the pattern, because there are three (or more) 4's, so the score returned is 4+4+1+4+4 = 17.

Example: An array containing the values 4, 4, 1, 4, 4 is passed to the **`scoreSmall`** function. The dice do not match the pattern of four consecutive values, so the score returned is zero.

You must implement all seven functions in your **yzscore.c** file. The definitions of your functions must match the declarations in **yzscore.h**. You may choose to implement additional functions.

User Interface

The user interface is implemented by the **main.c** file, so you don't need to implement it. Here's how it works.

The program prints "Roll: " and waits for the user to enter five integer values, separated by whitespace. The intent is that they are all entered on the same line, but it's not necessary – a linefeed is simply a whitespace character, so the integers can be on separate lines. But there may not be any other characters between the integers. (The results are undefined.)

The integers are checked (after all five are entered) to make sure they are between 1 and 6. If any of the values is illegal, the roll will be ignored and the prompt will be printed again.

Once a legal roll is read, the main function calls each of your functions in turn, and prints the score associated with each scoring category. Then the program loops back and prompts for another roll.

To end the program, the user types Ctrl-D. (This is the "end of file (EOF)" character.)

Example run:

```
% ./yzscore
Roll: 1 4 1 1 4
Three of a kind: 11
Four of a kind: 0
Full house: 25
Small straight: 0
Large straight: 0
Yahtzee: 0
Chance 11

Roll: 2 6 4 3 5
Three of a kind: 0
Four of a kind: 0
Full house: 0
Small straight: 30
Large straight: 40
Yahtzee: 0
Chance: 20

Roll: (user enters Ctrl-D)
%
```

Compiling

This time, we have two different source (**.c**) files to compile. (The header (**.h**) file is not compiled – it is included in the **.c** files when they are compiled.) We compile each source file separately, creating an object file (which has an extension of **.o**) for each. Then we link the object files together to create an executable file.

The **-c** flag is used to tell the compiler to create only an object file, not an executable. If you try to create an executable from either source file, the compiler (actually, the linker) will tell you that there are undefined functions. We need the functions from both object files to create a complete executable program.

Here are the steps:

1. Compile **main.c** file, creating **main.o**: `gcc -c -g -Wall -pedantic main.c`

2. Compile **yzscore.c** file, creating **yzscore.o**: `gcc -c -g -Wall -pedantic yzscore.c`

3. Link **main.o** and **yscore.o** to create an executable named **yzscore**:
```
gcc -o yzscore -g main.o yzscore.o
```

NOTE: Once you've done step (1), you shouldn't have to do it again, unless you delete **main.o**. You are not changing **main.c**, so there's no reason to recompile it. (Nothing will change!)

For NetBeans (or other IDE) users, your files are organized into a "project". You need to figure out how to add **main.c** and **yzscore.h** to your project. Then you create **yzscore.c**. When you tell the project to compile, it will automatically do the three steps above to create an executable.

I'd just "create" files with the appropriate names, and cut-and-paste in the code

Testing the Program

Try lots of different dice combinations, and make sure you get the correct score. Don't waste your time testing the user interface – this is not code that you're responsible for. If you find an error in the user interface, please tell us, but if you just want to make it "better," don't bother until you have finished implementing the scoring functions.

You may not change anything in **main.c** or **yzscore.h** to make your code work.

Do not assume that your functions will be called in any particular order. Just because **main.c** does things a certain way, this does not mean that our testing program will do things the same way. Each of your functions must work correctly on its own, independent of how or when it is called relative to the other functions. (NOTE: This means that if you set some global variable when **`score3`** is called, and use that value later in **`score4`**, this may not work in general. In our test program, we might call **`score4`** without calling **`score3`** first.)

Feel free to write your own testing code if you want, instead of relying on **main.c**.

If you want to implement and test one function at a time (good idea!!!), simply create "dummy" definitions of the other functions, which always return zero.

Hints and Suggestions

- Don't overcomplicate the program. Do not use anything that we have not covered in class.
- Read the Testing section (above) carefully. Each function must work independently of all other functions. Do not assume that they will be called in the same order as in the **main.c** code.
- Note that 1, 2, 3, 4, 5 is a large straight, but so is 5, 4, 3, 2, 1, and so is 2, 1, 4, 5, 3. In other words, the order of values doesn't matter. Consider creating another array that keeps track of how many 1's, 2's, 3's, etc., were rolled, independent of the order in which they appear. This will greatly simplify your code to match the dice against the desired patterns. (E.g., to find three-of-a-kind, look for any dice value that was counted three or more times.) There are other ways to do this, of course, but this is particularly useful for the small straight and large straight patterns.

- For compiler errors, look at the source code statement mentioned in the error. Try to figure out what the error is telling you. Try to fix the first error first, and then recompile. Sometimes, fixing the first error will make all the other errors go away. (Because the compiler got confused after the first error.)
- Use a source-level debugger, like *kdbg*, to step through the program if the program behavior is not correct. If you are using NetBeans on your own computer, the debugger is integrated with the editor and compiler, so there's no excuse for not using it.
- For general questions or clarifications, use the Message Board, so that other students can see your question and the answer. For code-specific questions, email your code (as an attachment) to the support list: ece209-001-sup@wolfware.ncsu.edu.

Email the instructor

Administrative Info

Updates or clarifications on the Message Board:

Any corrections or clarifications to this program spec will be posted on the Message Board. It is important that you read these postings, so that your program will match the updated specification.

Where to find files:

The **main.c** and **yzscore.h** files can be downloaded from the Programs web site, or can be copied from:

/afs/eos/lockers/workspace/ece/ece209/001/common/prog2

What to turn in:

- Source file – it must be named **yzscore.c**. Submit via Wolfware to the Program 2 assignment in your problem session section (201, 202, ..., 206). Your grade will be posted on the 209 gradebook, and you will be able to get detailed grading information using the "Retrieve Assignment" option on Wolfware.

Use moodle

Grading criteria:

Or -std=c99

15 points: File submitted with the proper name.

15 points: Compiles with no warnings and no errors (using –Wall and –pedantic). (If the program is not complete, then only partial credit is available here. In other words, you won't get 15 points for compiling a few trivial lines of code.)

10 points: Proper coding style, comments, and headers. No unnecessary global variables. No goto. (See the Programs web page for more information.)

10 points: Function scoreChance works correctly.

10 points: Function score3 works correctly.

10 points: Function score4 works correctly.

10 points: Function score5 works correctly.

10 points: Function scoreFull works correctly.

5 points: Function scoreLarge works correctly.

5 points: Function scoreSmall works correctly.

I'm going to be expecting more this time!