ECM2433 The C Family
Continuous Assessment

Date set: Monday 27th, February, 2023
Hand-in date: **12:00 Monday 27th March, 2023**

This continuous assessment (CA) comprises 30% of the overall module assessment.

This is an **individual** exercise and your attention is drawn to the University guidelines on collaboration and plagiarism, which are available from the University website. If you take inspiration for your code from another source (e.g. a web page or another student) then this **must be acknowledged** in the comments of your code.

This CA tests your knowledge of the programming in C, particularly the writing and testing of more complex programs.

Make sure that you lay your code out so that it is readable and you comment the code appropriately. All your functions should at least include a comment describing the arguments, what they return and what they do. You may like to adhere to one of the Doxygen conventions for this.

Your programs should compile and run using the `gcc` compiler on the `emps-ugcs1` and `emps-ugcs2` machines.

1 Pig Latin

Pig Latin is a simple made-up language that modifies English words, disguising their meaning to those who don't know the rules. See, for example, `http://en.wikipedia.org/wiki/Pig_Latin` for more on its history. According to Wikipedia, the rules for transforming an English word into a Pig Latin word are as follows:

1. For words that begin with consonants, the initial consonant or consonants are moved to the end of the word, and "ay" is added. For example,
   - happy $\Longrightarrow$ appyhay
   - duck $\Longrightarrow$ uckday
   - glove $\Longrightarrow$ oveglay
2. For words that begin with a vowel, the letters "way" are added at the end of the word. Thus
   - evil $\Longrightarrow$ evilway
   - eight $\Longrightarrow$ eightway
3. The letter "y" can play the role of either consonant or vowel. If it begins a word, then it acts as a consonant, but if it follows consonants, then it acts as a vowel. For example:
   - yowler $\Longrightarrow$ owleryay
   - crystal $\Longrightarrow$ ystalcray

   Write a function `char *pig(char *word)` that takes a *single* word as a string and returns a string containing the Pig Latin for the word. Write a program `test_pig.c` that tests each of the examples given above. Your test program should print each example, one per line, in the form:

   ```
   word => translation
   ```

for example

```
happy => appyhay
```

Use your function to write a program, `piglatin.c`, that repeatedly asks the user for a line of English and prints the corresponding Pig Latin translation. The program should stop when the user inputs an empty line. You may assume that the input lines do not contain punctuation.

You may find the C library functions `strsep` or `strtok` helpful. These functions will split a string into tokens (words); see the `man` pages for more information.

You should submit:

- A copy of your programs in the files `pig.c`, `test_pig.c` and `piglatin.c` together with any ancillary header files.
- A screenshot of your `piglatin` program in operation, showing the result of typing the phrases "The question of whether a computer can think is no more interesting than the question of whether a submarine can swim" and "My aunt yearns for yellow cats".

[20 marks]

2 Shuffling

It's often useful to be able to shuffle the items in a list. For example, in the next exercise it you will need to shuffle an array of integers representing a deck of cards. In this exercise you will write and test a function to shuffle the items in an array into random order. The idea is to emulate a "riffle shuffle" for shuffling cards. As illustrated in the pictures, in the riffle shuffle the cards are divided into two roughly equal-sized piles, one held in each hand with the thumbs held inward. They are then released so that the cards fall together interleaved. This process is repeated several times until the cards are in a random order.

Image from `www.wikipedia.org`

In a computer implementation of the riffle shuffle, the array to be shuffled will be divided into two halves, say A and B, and then reassembled into a new array. The first item in the new array will be the first item from one of the two half arrays, chosen with an equal probability; that is, there is a 50% chance that it is the first item of array A and 50% chance that it is the first item from array B. Likewise, each successive item in the shuffled array is, with equal probability, the next available item of A or B.

Write a procedure `riffle_once(void *L, int size, void *work)` which performs a single riffle shuffle of the array `L` each of whose elements is of size `size` bytes. It is surprisingly complicated to implement the riffle shuffle in place. Since we will need to repeat the shuffle several times it is inefficient to allocate additional memory each time `riffle_once` is called, so the array `work` should be an additional array of at least the same size as `L` that can be used as workspace.

Use `riffle_once` to write a new procedure `riffle(void *L, int size, int N)` which more thoroughly shuffles the array `L` by performing `N` successive riffles. The procedures `riffle_once` and `riffle` should be placed in the file `riffle.c`.

Demonstrate your code by writing a program, `demo_shuffle.c` that uses it to shuffle an array of integers $1, \ldots, 20$ and printing the result and then printing the result of shuffling the array of Greek letter names:

```
char *greek[] = "alpha", "beta", "gamma", "delta", "epsilon", "zeta","eta", "theta",
"iota", "kappa", "lambda", "mu"
```

Clearly, all the elements in the original array should be in the shuffled array and vice versa (why do we need to check both ways?). Write a function `int check_shuffle(void *L, int size)` that checks that your `riffle` function respects these conditions; it should return 1 if it does and 0 if not. This function should also be placed in `riffle.c`. Print the result of this check as part of your `demo_shuffle.c` program for the array of integers and the Greek letter names.

How many riffles are needed to properly shuffle the array? Clearly after a single riffle some items in the array that were originally next to or close to each other will still be close, but riffling many times will not make an already shuffled array any more shuffled. To answer this question we can define the quality of a shuffle as the fraction of times the second item of two adjacent items is larger than the first. Thus the sorted array {`0, 1, 2, 3, 4, 5, 6`} has a quality $6/6 = 1$ because every item after the first is greater than the previous one; that is $1 > 0$, $2 > 1$, $3 > 2$, $4 > 3$, $5 > 4$ and $6 > 5$. On the other hand, the array {`1, 4, 2, 3, 6, 5, 0`} has a quality $3/6 = 0.5$ because the pairs $(1, 4), (2, 3), (3, 6)$ have the second element greater than the first, but the other pairs $(4, 2), (6, 5), (5, 0)$ do not. A well-shuffled array will have a quality close to 0.5.

Write a function `float quality(int *numbers)` that evaluates how well an array of integers `numbers` is shuffled. Then write a function `average_quality(N, trials)` that uses these functions to evaluate the average quality of a shuffle of the integers `0, 1, ..., N-1`, using `riffle` to shuffle the array `N` times. To get reliable answers you will need to take the average over, say, 30 different `trials`. Finally, write a program that prints out the average quality of a riffle shuffle of an array of length 50 using $N = 1, 2, 3, \ldots, 15$. Based on your results, how big do you think N should be to properly shuffle a list? Your `quality` and `average_quality` functions should be contained in `riffle.c` and the driver program in `quality.c`.

You should submit:

- Code in the files `riffle.c`, `demo_shuffle.c` and `quality.c` together with any required header files.
- A text file, `quality.txt`, that contains the output of your `quality.c` program and an explanation of how many times you think it is necessary to shuffle an array of length 50 to achieve a good shuffle.

[35 marks]

3 Beggar your neighbour

In the card game beggar-your-neighbour, a standard 52-card deck is dealt to the players. Each player places their cards face down on the table. The first player lays down her top card face up, and the next player plays her top card, face up, on it. This continues for each player in turn until a penalty card (Jack, Queen, King or Ace) is laid.

If a player plays a penalty card the next player has to pay a penalty by laying down additional cards: 4 cards for an Ace, 3 for a King, 2 cards for a Queen and 1 for a Jack. When the penalty has been paid the player who laid the penalty card takes all the cards in the pile and places them under her cards. Play continues with the player following the player who picked up the cards, who lays a new card face up. However, if the player who is paying the penalty turns up a penalty card herself, then her payment ceases and the next player in turn must pay the penalty indicated by the new penalty card. A player leaves the game when they have no cards left and the game is won by the player who eventually has all the cards.

Here is an excerpt from a game with 3 players. The cards held by each player are represented by integers $2, 3, \ldots, 14$ with $11 =$ Jack, $12 =$ Queen, $13 =$ King and $14 =$ Ace. The `*` indicates the player whose turn it is and 'Pile' means the cards that have been laid on the table.

```
Turn 21                       Top card in pile is 9, so player 0 should lay a single card
Pile:  10, 10, 4, 6, 4, 8, 6, 5, 7, 9
*   0: 4, 10, 9, 3, 3, 2, 12, 8, 9, 2, 12, 4, 11, 11, 2, 3, 11, 5, 7, 12, 5, 3
    1: 13, 6, 9, 8, 2, 6, 10, 7
    2: 13, 8, 13, 5, 11, 13, 7, 12

Turn 22                       Top card in pile is 4, so player 1 should lay a single card
Pile:  10, 10, 4, 6, 4, 8, 6, 5, 7, 9, 4
    0: 10, 9, 3, 3, 2, 12, 8, 9, 2, 12, 4, 11, 11, 2, 3, 11, 5, 7, 12, 5, 3
*   1: 13, 6, 9, 8, 2, 6, 10, 7
    2: 13, 8, 13, 5, 11, 13, 7, 12

Turn 23                            Top card is 13, so player 2 should lay 3 penalty cards
Pile:  10, 10, 4, 6, 4, 8, 6, 5, 7, 9, 4, 13
    0: 10, 9, 3, 3, 2, 12, 8, 9, 2, 12, 4, 11, 11, 2, 3, 11, 5, 7, 12, 5, 3
    1: 6, 9, 8, 2, 6, 10, 7
*   2: 13, 8, 13, 5, 11, 13, 7, 12

Turn 24 First card laid by player 2 was a penalty card, 13, so now player 0 to lay 3 cards
Pile:  10, 10, 4, 6, 4, 8, 6, 5, 7, 9, 4, 13, 13
*   0: 10, 9, 3, 3, 2, 12, 8, 9, 2, 12, 4, 11, 11, 2, 3, 11, 5, 7, 12, 5, 3
    1: 6, 9, 8, 2, 6, 10, 7
    2: 8, 13, 5, 11, 13, 7, 12

Turn 25
        Player 0 laid 10, 9, 3 without laying a penalty card, so the pile at the
        end of last turn is given to player 2.
        Pile now empty so player 1 to lay a single card
Pile:
    0: 3, 2, 12, 8, 9, 2, 12, 4, 11, 11, 2, 3, 11, 5, 7, 12, 5, 3
*   1: 6, 9, 8, 2, 6, 10, 7
    2: 8, 13, 5, 11, 13, 7, 12, 10, 10, 4, 6, 4, 8, 6, 5, 7, 9, 4, 13, 13, 10, 9, 3

Turn 26                                                     Player 2 to lay a single card
Pile: 6
    0: 3, 2, 12, 8, 9, 2, 12, 4, 11, 11, 2, 3, 11, 5, 7, 12, 5, 3
    1: 9, 8, 2, 6, 10, 7
*   2: 8, 13, 5, 11, 13, 7, 12, 10, 10, 4, 6, 4, 8, 6, 5, 7, 9, 4, 13, 13, 10, 9, 3
```

Write a function `int beggar(int Nplayers, int *deck, int talkative)` to play a *single* game of beggar-your-neighbour with `Nplayers` using the shuffled `deck` of cards. If the argument `talkative` is not zero, your function should print details of what it is doing, similar to the excerpt above. Your function should return the number of turns that it takes for the game to finish.

- Consider how you will represent the cards held by a player. Note that cards are taken in order from one end of the set of cards the player holds and added to the other end.
- You should write a function `finished(players)` that returns true (non-zero) if one player has all the cards and the others have none.
- You should write a function `reward = take_turn(&player, &pile)` that takes the cards held by the `player` whose turn it is and the current `pile` of cards. It should update the player's cards and the pile after the turn and return, as the `reward`, the cards that should be given to the previous player because this player didn't lay another penalty card in response to the top card of the pile. Thus in Turn 24 above, the reward would be the cards `10, 10, 4, 6, 4, 8, 6, 5, 7, 9, 4, 13, 13, 10, 9, 3`. Often the `reward` will be empty and if `reward` is not empty, then the updated pile will be empty.

- You can use your `riffle` function from the previous question to shuffle the deck, but it might be more reliable to use the shuffle function in `shuffle.c` on the ELE page which wraps the GNU Scientific Library to shuffle an array of integers *in place*. To use this you will need to link your code with the GNU Scientific Library using `-lgsl`.
- This code should be in a file `beggar.c`. Write a program `single.c` that takes the number of players as a command line argument and prints the details of a game played with that many players.

It might be possible that a game of beggar-your-neighbour goes on forever because the cards keep being passed back and forth between the players. Whether an infinite game exists is an open (but not very important) question. Nonetheless, write a function `stats = statistics(int Nplayers, int games)` that uses your `beggar` function to find the shortest, average and longest of `games` games with `Nplayers`. It should return a `strut` with fields `shortest`, `longest` and `average`. Use this function in a program (also in `byn.c`) to print these statistics for games with $2, 3, 4, \ldots, 10$ players using at least 100 trials each. Your `byn` program should take two command line arguments: the maximum number of players and the number of trials.

You should submit:

- Your `beggar.c`, `single.c` and `byn.c` files together with any ancillary source and header files. You should also provide a `Makefile` that can be used to compile the `single` and `byn` programs.
- Output of your `byn` program in a file `statistics.txt`.

[45 marks]

[Total 100 marks]

Marking criteria

Work will be marked against the following criteria. Although it varies a bit from question to question they all have approximately equal weight.

- **Does your algorithm correctly solve the problem?**
  In most of these exercises the algorithm has been described in the question, but not always in complete detail and some decisions are left to you.
- **Does the code correctly implement the algorithm?**
  Have you written correct code? Do your functions meet the specification?
- **Is the code syntactically correct?**
  Is your program a legal C program regardless of whether it implements the algorithm?
- **Is the code beautiful?**
  Is the implementation clear and efficient or is it unclear and inefficient? Have you used appropriate data structures? Is the code well structured? Have you made good use of functions?
- **Is the code well laid out and commented?**
  Is there a comment describing what the code does? Are there comments describing the major portions of the code or particularly tricky bits? Do functions have a comment describing what they do and how they are used? Although C doesn't care about indentation, have you used indentation and space to make the code clear to human readers?

There is a **10% penalty for not naming files** as instructed in the questions.