VE 280 Lab 3

Out: May 31, 2020; **Due**: 11:59 pm, June 7, 2022.

Ex1. Card Game!

Related Topics: *Recursion*.

Problem: Let's play with cards! You are given a sequence of `n` cards, all placed face down, and the number written on the other side of each card is guaranteed to be unique and positive. Each card is also indexed from 0 to `n-1`, which corresponds to its position in the sequence. Among the cards, there is always one with **280** on it. The rule of the game is stated as follows:

1. You will be given a start position, namely *p*. Make its face up.
2. If the card on that position has value **280**, then you win. Otherwise, proceed to step 3.
3. Suppose the value on the card at this position *p* is *v*. Then you can choose **either** the card on position *p+v* **or** position *p-v*. Once a card is chosen, **make its face up**. Note that the chosen position must be **valid**, i.e., you cannot choose a card outside the given sequence. Moreover, you cannot choose a card already face up. If no card can be chosen, you lose.
4. Once you've picked your card from step 3, update the current position, go to step 2.

Requirement: Please refer to `ex1.cpp` in the `starter_files` folder. The skeleton code is already provided. You are required to use **recursion** to implement the function below, where `count` is the number of element in the card array, `arr[]` represents the sequence of values on each card, `position` is the start position of the game. You may use a helper function if you want.

```
bool canWin(int count, int arr[], int position) {
    // TODO: implement this function
}
```

Input Format: There will be 3 lines, the first line only contains the number of cards in the sequence. The second line gives the values of all the cards. The third line specifies the start position (0-index based).

```
// sample input
3
280 203 281
0
```

Output Format: Either `0` (for false) or `1` (for true).

```
1
```

Hint:

1. We guarantee that the start position is valid.
2. Once a card has been chosen, you cannot choose it again. Therefore, you should find a way to record this.

Ex2. Fold And Fold Again

Related Topics: *Recursion*, *Function pointer*.

In this exercise, you will need to implement the fold function below, where `count` is the number of elements in the `arr` array, `fn` points to a function, which takes in two `int`s and returns an `int`, and `initial` is a value. You are welcomed to write a helper function for the recursion.

```
int fold (int count, int arr[], int (*fn) (int, int), int initial) {

}
```

If `count == 0`, just return the `initial` value. Otherwise, take `count = 3` for example, the return value could have been equivalently calculated as follows:

```
int temp1 = fn(arr[0],initial)
int temp2 = fn(arr[1],temp1)
int temp3 = fn(arr[2],temp2)
return temp3
```

In short, the `fold` function will take each element in the array as well as the previous return value, until we have gone through the whole array. And it will return the last calculated value.

To help you understand how the `fold` function works, write two `fn` functions to be used with the `fold` function:

- a function called `fn_add` such that `fold(n, arr, fn_add, 0)` returns the sum of the elements in `arr`.
- a function called `fn_count_odd` such that `fold(n, arr, fn_count_odd, 0)` returns the number of odd numbers in `arr`.

Hint: Please be aware that the order of inputs into the `fn` function matters. The first one should be the element from the array, and the second one should be either the initial value or the previous return value of `fn`.

Requirement: Please refer to `ex2.cpp` in the `starter_files` folder, you will need to implement `fold`, `fn_add` and `fn_count_odd` functions. You only need to submit `ex2.cpp` for this problem.

Ex.3 Play with Integers

Related Topics: *Program Arguments*.

Problem: You will use `cin` to read an array of `int` and do some data manipulation according to the program arguments. There are several options for the program:

- --help: print the message `Hey, I love Integers.`
  and exit, ignore all other options.
- --add: add all the numbers in the array, record the result. No need to worry about the integer overflow.
- --small `<int>`: find the smallest element in the array, add this element to the `int` passed through the argument, and print the result.
- --verbose: verbose mode, see the output format for details.

Input Format:
There will be 2 lines, the first line only contains the number of `int` in the array. The second line gives the whole array.

```
// sample input
5
1 2 3 4 5
```

Example 1 (contains `--help`):

Program Argument: `./ex3 --help` or `./ex3 --verbose --help`

Output Format: `Hey, I love Integers.` Notice that when we have `--help` in the argument, we will only print this line and return.

Example 2 (contains `add`):

Program Argument: `./ex3 --add`

Output Format: `15` because `1+2+3+4+5=15`.

Example 3 (contains `verbose`):

Program Argument: `./ex3 --add --verbose`

Output Format:

```
This is add operation.
15
```

Example 4 (contains `verbose`):

Program Argument: `./ex3 --small 3 --verbose`

Output Format:

```
This is small operation.
4
```

Note that if no data operation is mentioned in the argument, no matter whether the verbose mode is on or not, we simply output one line: `No work to do!`. If both data operations are there, always first output the things related to the **add** operations. Please also be aware that the order of all the options will be **random** and you may assume that there are no unknown options in the argument and the argument must be valid.

Requirement: Create a file with name `ex3.cpp` and write your function there.

Submission:

Compress all **.cpp** file into a **.tar** or **.zip** file and submit to JOJ.

Do not change the name of any **.cpp** files.
