COMP 322
Winter Semester 2022

INSTRUCTOR: DR. CHAD ZAMMAR
chad.zammar@mcgill.ca

Assignment 1: Exploring Functions and arrays.

Due date: 11 February 2022, 11:59 PM.

Before you start:

- Research for similar problems on the internet is recommended. However, your submission should reflect individual work and personal effort.
- Some of the topics may not be covered in class due to our limited time. You are encouraged to find answers online. You can also reach out to the TAs for guidance.
- Please submit your assignment before the due date to avoid penalties or worse risking your assignment being rejected.
- Submit one file called **assignment1.cpp** containing all the functions together with their implementations. It will also contain the main() function that runs everything.

Make sure your code is clear and readable. **Readability of your code as well as the quality of your comments will be graded**.

- No submission by email. Submit your work to mycourse.
- If your code does not compile it will not be graded.
- Be happy when working on your assignment, because a happy software developer has more inspiration than a sad one :).

C++ offers a myriad of containers to store objects. The most basic one is the array. Instead of storing a single value in a variable we can store multiple values at once in an array. **For this assignment we will be using the basic c-style array and not any of the more advanced C++ containers such as vectors etc.**

The operating system manages the memory in an array-like fashion and hides the complexities from users. In this assignment we will be imitating the inner workings of the operating system by managing a piece of memory of a fixed size and then enabling clients to use this memory without collision between users. We will be building a Data Store manager to help us get access to chunks of memory that we can use to store integers.

To make your task easier, a C++ skeleton code is provided for you (check assignment1.cpp file).

Our data store is modeled after a c-style array of fixed size that can hold a maximum of 50 integer elements. We can define it this way

```
1  #define ARRSIZE 50
2  int datastore[ARRSIZE] = {};
```

Note that datastore is initialized to 0. A client should not interact directly with **datastore** array but through an interface (meaning a collection of functions) that you will be implementing.

Please note that 10 points will be given for code readability and quality of comments provided.

Question 1 [10 points]

The first function to implement is a function that the client will call to book a data store of size *ssize*. Let's call this function newstore.

```
int newstore(int ssize)
{
    //
}
```

newstore function takes the store size as input argument and returns a generated integer number which represents the ID of the newly created store.

For example, in the following code snippet we are creating 2 separate data stores. The first one is called s1 and it is calling the function newstore in order to initialize a new data store of size 3 (meaning it can hold 3 integers). The second call is creating a new data store called s2 that can hold a total of 5 integers.

```
int s1 = newstore(3); // create new empty data store of size 3
int s2 = newstore(5); // create new empty data store of size 5
```

The return value of the newstore function is a generated number that start at 0 and will be incremented by one after each call. s1 will hold the value 0 if the call for newstore was successful, -1 otherwise. s2 will hold the value 1 if the call was successful, -1 otherwise.

What does it mean for **newstore** call to be successful or not? It means that, if we have enough empty space in our datastore array to accommodate the demand, then the call should be successful. The call should fail otherwise, meaning that if datastore was already fully booked, we should not permit overbooking and therefore the return value of the newstore function should be -1 indicating that there is no more available space.

Question 2 [10 points]

We now need a function to enable users adding elements to the store. Let's call this function *add_element_to_store*

```
int add_element_to_store(int id, int val, int idx=-1)
{
    // if idx was not provided by user, then append value at the end
}
```

The inputs are as follow:

id: the store id to which we want to add an element

val: the value that we want to add

idx: the local index where we want to insert the element. If idx was not provided by the user, append the new element at the end of the store given by id.

The return value should be zero if the operation was successful, -1 otherwise (in case we run out of capacity or if the index provided is out of range).

For example:

```
int s1 = newstore(3); // create new empty data store of size 3
int s2 = newstore(5); // create new empty data store of size 5

if (s1 != -1)
{
    add_element_to_store(s1, 13);
}
```

We are adding the value 13 to the store s1. Idx was not provided, meaning that the value should be appended at the end of s1.

If idx was provided, then you need to make sure that idx's value does not exceed the total size of s1. If the provided idx was outside the acceptable range for s1, the functions should return -1 and the element will not be added to s1.

For example:

add_element_to_store(s1, 13, 1) // this is OK. insert 13 at the second position

add_element_to_store(s1, 13, 2) // this is OK. insert 13 at the third position

add_element_to_store(s1, 13, 3) // this is NOT OK. 4th position exceeds the capacity of s1 which is 3, so the function should return -1.

Please note how we tested the value of s1 before trying to add anything to it to make sure that the store was created first.

Question 3 [10 points]

Now let's create a debug function to help us track the evolution of datastore and to also make debugging easier. The prototype of the debug function is as follow:

```
1  void print_debug()
2  {
3      //
4  }
```

You can use print_debug after every operation to make sure that the system is behaving as expected. The output of print_debug is simply the count of the available elements in the data store, the content of the whole datastore array and the individual content of every data store that was being initialized so far.

For example, if you run print_debug before using the system, it should reflect the status of the system as it is initially:

```
1  int main()
2  {
3      print_debug();
4  }
```

This should print the following:

```
available elements in datastore: 50
datastore   : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
####################################
```

Try calling the function again after adding some elements to s1:

```
int main()
{
    print_debug();
    int s1 = newstore(3); // create new empty data store of size 3
    int s2 = newstore(5); // create new empty data store of size 5

    if (s1 != -1)
    {
        add_element_to_store(s1, 13);
    }
    print_debug();
}
```

The output will be as follow:

```
available elements in datastore: 50
datastore   : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
####################################
available elements in datastore: 42
datastore   : 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 0: 13 0 0
store 1: 0 0 0 0 0
####################################
```

Let's add another element to s2 then print_debug again:

```
int main()
{
    print_debug();
    int s1 = newstore(3); // create new empty data store of size 3
    int s2 = newstore(5); // create new empty data store of size 5

    if (s1 != -1)
    {
        add_element_to_store(s1, 13);
    }
    print_debug();
    if (s2 != -1)
    {
        add_element_to_store(s2, 7, 2);
    }
    print_debug();
}
```

The output would be:

```
available elements in datastore: 50
datastore   : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
###################################
available elements in datastore: 42
datastore   : 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 0: 13 0 0
store 1: 0 0 0 0 0
###################################
available elements in datastore: 42
datastore   : 13 0 0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 0: 13 0 0
store 1: 0 0 7 0 0
###################################
```

Please note here that we added 7 at idx=2, meaning at the third position in store s1. idx is local to the store s1and not to the global datastore.

Question 4 [10 points]

We need to give the user the possibility of removing an element from its store either by index or by value. Let's implement these 2 functions:

```
1  void delete_element_from_store_by_value(int id, int val)
2  {
3      // delete first occurence of val from store id
4  }
5
6  void delete_element_from_store_by_index(int id, int idx)
7  {
8      // delete element at index idx from store id
9  }
```

Deleting an element simply sets its value to zero. Deleting an element by value means that we need to set the value of the first occurrence of an element to zero. If the element was not found, it will be ignored and nothing is deleted.

Deleting by index means that we need to set the value located at the specified index to zero. The index idx is local to the store given by id and not global to datastore. If idx was out of range, it should be ignored and nothing is deleted.

The following code:

```
int main()
{
    int s1 = newstore(3); // create new empty data store of size 3
    int s2 = newstore(5); // create new empty data store of size 5

    if (s1 != -1)
    {
        add_element_to_store(s1, 13);
        add_element_to_store(s1, 15);
        add_element_to_store(s1, 21);
    }

    if (s2 != -1)
    {
        add_element_to_store(s2, 7, 2);
    }
    print_debug();
    delete_element_from_store_by_value(s1, 15);
    print_debug();
    delete_element_from_store_by_index(s2, 2);
    print_debug();
}
```

Should print the following result:

```
available elements in datastore: 42
datastore   : 13 15 21 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 0: 13 15 21
store 1: 0 0 0 0 0
#####################################
available elements in datastore: 42
datastore   : 13 0 21 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 0: 13 0 21
store 1: 0 0 7 0 0
#####################################
available elements in datastore: 42
datastore   : 13 0 21 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 0: 13 0 21
store 1: 0 0 0 0 0
#####################################
```

Question 5 [15 points]

Implement the following function that given a value as input, will print to the screen the list of all the store IDs having this value:

```
void which_stores_have_element(int val)
{
    // print the store ids for all the stores that have the value val
}
```

For example:

```
int main()
{
    int s1 = newstore(3); // create new empty data store of size 3
    int s2 = newstore(5); // create new empty data store of size 5

    if (s1 != -1)
    {
        add_element_to_store(s1, 13);
        add_element_to_store(s1, 15);
        add_element_to_store(s1, 21);
    }

    if (s2 != -1)
    {
        add_element_to_store(s2, 7, 2);
        add_element_to_store(s2, 15, 0);
        add_element_to_store(s2, 22, 1);
    }
    print_debug();
    which_stores_have_element(15);
    which_stores_have_element(22);
    which_stores_have_element(48);

}
```

Will give the following output:

```
available elements in datastore: 42
datastore   : 13 15 21 15 22 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 0: 13 15 21
store 1: 15 22 7 0 0
#####################################
Element 15 is in store IDs: 0, 1
Element 22 is in store IDs: 1
Element 48 is not available in any store
```

Question 6 [15 points]

Implement a function that will delete a store:

```
void delete_store(int id)
{
    // shift all stores and update the number of total available elements
}
```

When deleting a store, we need to make sure that all its elements are set to 0 and that the number of available elements is updated accordingly. You need to shift all the stores to the left in datastore array to avoid creating a fragmented memory.

For example:

```c
int main()
{
    int s1 = newstore(3); // create new empty data store of size 3
    int s2 = newstore(5); // create new empty data store of size 5

    if (s1 != -1)
    {
        add_element_to_store(s1, 13);
        add_element_to_store(s1, 15);
        add_element_to_store(s1, 21);
    }

    if (s2 != -1)
    {
        add_element_to_store(s2, 7, 2);
        add_element_to_store(s2, 15, 0);
        add_element_to_store(s2, 22, 1);
    }
    print_debug();
    which_stores_have_element(15);
    which_stores_have_element(22);
    which_stores_have_element(48);

    delete_store(s1);
    print_debug();

}
```

Will give the following output:

```
available elements in datastore: 42
datastore   : 13 15 21 15 22 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 0: 13 15 21
store 1: 15 22 7 0 0
#####################################
Element 15 is in store IDs: 0, 1
Element 22 is in store IDs: 1
Element 48 is not available in any store
available elements in datastore: 45
datastore   : 15 22 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 1: 15 22 7 0 0
#####################################
```

Question 7 [20 points]

A user may realize later that he needed more space (or less space) for his store than expected. We need to provide a way to make a store resizable. Implement the following function that will check first if we still have enough space to accommodate the new demand. If so, then you need to update the space requirement for the store given by its id. This operation may require you to shift all the other stores in datastore. Feel free to find a suitable implementation. Remember to update the total number of available elements in datastore.

```
1  int resize_store(int id, int newsize)
2  {
3      // returns -1 if resizing was not successful. 0 otherwise.
4  }
```

For example:

```c
int main()
{
    int s1 = newstore(3); // create new empty data store of size 3
    int s2 = newstore(5); // create new empty data store of size 5

    if (s1 != -1)
    {
        add_element_to_store(s1, 13);
        add_element_to_store(s1, 15);
        add_element_to_store(s1, 21);
    }

    if (s2 != -1)
    {
        add_element_to_store(s2, 7, 2);
        add_element_to_store(s2, 15, 0);
        add_element_to_store(s2, 22, 1);
    }
    print_debug();

    resize_store(s1, 7);

    print_debug();

}
```

Will produce the following output:

```
available elements in datastore: 42
datastore   : 13 15 21 15 22 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 0: 13 15 21
store 1: 15 22 7 0 0
#####################################
available elements in datastore: 38
datastore   : 13 15 21 0 0 0 0 15 22 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 0: 13 15 21 0 0 0 0
store 1: 15 22 7 0 0
#####################################
```

Please use the following main() function for testing:

```
int main()
{
    int s1 = newstore(3); // create new empty data store of size 3
    int s2 = newstore(5); // create new empty data store of size 5

    if (s1 != -1)
    {
        add_element_to_store(s1, 13);
        add_element_to_store(s1, 15);
        add_element_to_store(s1, 21);
        add_element_to_store(s1, 42); // this should return -1
    }

    if (s2 != -1)
    {
        add_element_to_store(s2, 7, 2);
        add_element_to_store(s2, 15, 0);
        add_element_to_store(s2, 22, 1);
    }
    print_debug();

    delete_element_from_store_by_value(s1, 13);
    delete_element_from_store_by_value(s1, 71);
    delete_element_from_store_by_index(s2, 2);
    delete_element_from_store_by_index(s1, 5);
    print_debug();

    which_stores_have_element(15);
    which_stores_have_element(29);

    delete_store(s1);
    print_debug();

    resize_store(s2, 20);
    int s3 = newstore(40);
    print_debug();

    s3 = newstore(30);
    add_element_to_store(s3, 7, 29);
    print_debug();
```

```
}
```

When you run the provided main() function, you should get the following output to the screen:

```
available elements in datastore: 42
datastore   : 13 15 21 15 22 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 0: 13 15 21
store 1: 15 22 7 0 0
#####################################
available elements in datastore: 42
datastore   : 0 15 21 15 22 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 0: 0 15 21
store 1: 15 22 0 0 0
#####################################
Element 15 is in store IDs: 0, 1
Element 29 is not available in any store
available elements in datastore: 45
datastore   : 15 22 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 1: 15 22 0 0 0
#####################################
available elements in datastore: 30
datastore   : 15 22 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 1: 15 22 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
#####################################
available elements in datastore: 0
datastore   : 15 22 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7
store 1: 15 22 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
store 2: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7
#####################################
```

Do not modify the main() function and do not implement any logic in it. All your code should be provided within the provided functions. main() function will be used for testing only.