

Exercise 2.1

ex2-1.c

```
1 #include <stdio.h>           // include stdio to get commands like printf
2
3 int main()
4 {
5     int len = 10;             // Used to set the length of the array
6     int arr[len];            // Signed integer array of size len
7     int *arr_ptr = &arr[0];   // Pointer to the first element of the array
8
9     // Fill the array with integers from 0 to 9
10    for (int i = 0; i < len; i++)
11    {
12        arr[i] = i;
13    }
14
15    // Print the array using the [ ] syntax
16    printf("%s", "Array in reverse Order using [ ] : \0");
17    for (int i = len-1; i >= 0; i--)
18    {
19        printf("%d, ", arr[i]);
20    }
21
22    printf("%s", "\n\0"); // new line
23
24    // Print the array using pointer arithmetic
25    printf("%s", "Array in reverse Order using a pointer: \0");
26    for (int i = len-1; i >= 0; i--)
27    {
28        printf("%d, ", *(arr_ptr + i));
29    }
30
31    printf("%s", "\n\0"); // new line
32
33    return 0; // indicates that program exited successfully
34 }
```

We compile the program with: `gcc -o ex2-1 ex2-1.c` and receive the compiled executable `ex2-1` which we can run using `./ex2-1`.

Console Output:

```
Array in reverse Order using [ ] : 9, 8, 7, 6, 5, 4, 3, 2, 1, 0,
Array in reverse Order using a pointer: 9, 8, 7, 6, 5, 4, 3, 2, 1, 0,
```

Exercise 2.2

ex2-2.c

```
1 #include <stdio.h>
2
3 // Set the "blueprint" on how a Vector looks like
4 struct Vector
5 {
6     int a;
7     int b;
8     int c;
9 };
10
11 // Method that adds vector 'b' to vector 'a'
12 void addVec(struct Vector *a, struct Vector *b)
13 {
14     a->a += b->a;
15     a->b += b->b;
16     a->c += b->c;
17 }
18
19 // Method to print a vector
20 void printVec(struct Vector *a)
21 {
22     printf("(%d, %d, %d)\n", a->a, a->b, a->c);
23 }
24
25
26 int main()
27 {
28     struct Vector a = {1, 5, 3};      // initialize two vectors
29     struct Vector b = {1, 2, 3};
30
31     addVec(&a, &b);                // add vector b to vector a
32                             // a should now be (2, 7, 6)
33
34     printVec(&a);                  // print
35
36     return 0;
37 }
```

We compile the program with: `gcc -o ex2-2 ex2-2.c` and receive the compiled executable `ex2-2` which we can run using `./ex2-2`.

Console Output:
(2, 7, 6)

Exercise 2.3

First we include `stdio.h` and `stdlib.h` to get access to `printf()`, `malloc()` and `free()`. Then we define our `Vector` struct. We could also include it from the last exercise but then the *two* main functions will do trouble. For the linked list we need a `Node` structure to hold the data and a pointer to the next Node and a List Structure (`VectorList`) that holds pointers to the head of the list and optionally also to the tail.

Additionally we write a short helper function `printVec` that helps us to easily print our vectors.

Because the `size()` method is used before it is defined, we write this prototype here to let the compiler know of its existence.

ex2-3.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 // Defining structs
5 struct Vector
6 {
7     int a;
8     int b;
9     int c;
10};
11
12 struct Node
13 {
14     struct Node *next;
15     struct Vector *vector;
16};
17
18 struct VectorList
19 {
20     struct Node *head_ptr;
21     struct Node *tail_ptr;
22};
23
24 int size(struct VectorList *l); // function prototype
25
26 // Helper method to print a vector
27 void printVec(struct Vector *a)
28 {
29     printf("(%d, %d, %d)\n", a->a, a->b, a->c);
30}
```

Now we're writing the first method `getElement()` which returns a pointer to the `Vector` at the given `index` in the `VectorList`. To do that, we first check that the given `index` is not out of bounds. Secondly we handle the case when the `VectorList` is empty. With those edge-cases handled, we either return `head_ptr->vector` when the index is 0 or we iterate through the list to the requested `index`.

ex2-3.c

```
32 // Returns pointer to vector at index
33 struct Vector* getElement(struct VectorList *l, int index)
34 {
35     if (index < 0 || index > size(l))
36     { // Check if index is out of bounds
37         printf("Index is out of bounds! Index: %d", index);
38         return NULL;
39     }
40
41     if (l->head_ptr == NULL)
42     { // Check if list is empty
43         printf("The list is empty.");
44         return NULL;
45     }
46
47     if (index == 0)
48     { // First Element
49         return l->head_ptr->vector;
50     }
51
52     // Iterating through the list
53     struct Node *curr = l->head_ptr;
54     for (int i = 0; i < index; i++)
55     {
56         curr = curr->next;
57     }
58     return curr->vector;
```

The `insertElementFront()` method is supposed to add an element to the front of the `Vectorlist`. Since the list should be stored on the heap, we need to allocate space for a new `Node` where we can store the `Vector` pointer in. We initialize a temporary pointer to point at this new `Node` in the heap, store the `vector` pointer inside it and let it point to the head of the `VectorList`. Therefore making it the new head. We change the `head_ptr` which currently points to the second element to now point to the first. Lastly we take care of an edge-case, if the `VectorList` was empty before, its `tail_ptr` also has to point to our newly added `Node` because with only a single element, tail and head are the same.

ex2-3.c

```
62 void insertElementFront(struct VectorList *l, struct Vector *v)
63 {
64     // Allocate a space for a node structure on the heap with a tmp_ptr
65     // pointing to it.
66     struct Node *tmp_ptr = (struct Node*) malloc(sizeof(struct Node));
67
68     // Insert the passed vector to that node and connect it with the head
69     tmp_ptr->next = l->head_ptr;
70     tmp_ptr->vector = v;
71
72     // Change the head_ptr to point to the new head
73     l->head_ptr = tmp_ptr;
74
75     // When list was empty, set the tail_ptr also to the new node.
76     if (l->tail_ptr == NULL)
77     {
78         l->tail_ptr = tmp_ptr;
79     }
}
```

The next method does basically the same, just this time adding the element to the back instead of the front. We again create a `Node` on the heap with a temporary pointer pointing to it. This time though the new `Node` points to `NULL` since it is the last `Node`.

We then handle the case of the empty `VectorList` where our last `Node` is also the first one. We therefore let the `head_ptr` point to it. On the other hand, if the `VectorList` was not empty, its current last `Node` needs to point to our new node. And finally we also let the `tail_ptr` point to our new last `Node`.

ex2-3.c

```
81 void insertElementBack(struct VectorList* l, struct Vector* v)
82 {
83     // Allocate a space for a node structure on the heap with a tmp_ptr
84     // pointing to it.
85     struct Node *tmp_ptr = (struct Node*) malloc(sizeof(struct Node));
86
87     // Insert the passed vector to that node.
88     tmp_ptr->next = NULL;
89     tmp_ptr->vector = v;
90
91     // If list was empty, let the head point to the new node.
92     if (l->tail_ptr == NULL)
93     {
94         l->head_ptr = tmp_ptr;
95     }
96     else // connect the last node with the new last node.
97     {
98         l->tail_ptr->next = tmp_ptr;
99     }
100    l->tail_ptr = tmp_ptr; // set the new tail
101
102 }
```

The `size()` method simply iterates through the list until the current `Node` is `NULL` and then returns the counter.

ex2-3.c

```
104 int size(struct VectorList *l)
105 {
106     struct Node *curr = l->head_ptr;
107     int count = 0;
108     while (curr != NULL) {
109         curr = curr->next;
110         count += 1;
111     }
112     return count;
113 }
```

In the `main()` method we create a `VectorList` and a pointer to it and some `Vectors`, also with pointers to them. We're then inserting the `Vectors` in our `VectorList` using the two different methods and then print out its size as well as the elements using the `getElement()` method.

ex2-3.c

```
115 int main() {
116     struct VectorList list = {NULL, NULL};
117     struct VectorList *l_ptr = &list;
118
119     struct Vector a = {1, 1, 1};
120     struct Vector b = {2, 2, 2};
121     struct Vector c = {3, 3, 3};
122     struct Vector d = {4, 4, 4};
123     struct Vector *a_ptr = &a;
124     struct Vector *b_ptr = &b;
125     struct Vector *c_ptr = &c;
126     struct Vector *d_ptr = &d;
127
128     printf("Size of the list is %d\n", size(l_ptr));
129     insertElementFront(l_ptr, b_ptr);
130     insertElementBack(l_ptr, c_ptr);
131     insertElementBack(l_ptr, d_ptr);
132     insertElementFront(l_ptr, a_ptr);
133
134     printf("Size of the list is %d\n", size(l_ptr));
135     printf("First element is:\n");
136     printVec(getElement(l_ptr, 0));
137     printf("Second element is:\n");
138     printVec(getElement(l_ptr, 1));
139     printf("Third element is:\n");
140     printVec(getElement(l_ptr, 2));
141     printf("Fourth element is:\n");
142     printVec(getElement(l_ptr, 3));
143     return 0;
144 }
```

We compile the program with: `gcc -o ex2-3 ex2-3.c` and receive the compiled executable `ex2-3` which we can run using `./ex2-3`. Console Output:

```
Size of the list is 0
Size of the list is 4
First element is:
(1, 1, 1)
Second element is:
(2, 2, 2)
Third element is:
(3, 3, 3)
Fourth element is:
(4, 4, 4)
```

Exercise 2.4

Just like before we create our `Vector` struct, a helper function to print the vector, another one to print an array of vectors and a third one to calculate the L2 form of a vector.

Normally we could just `#include` those things from the other files but since every file has its own `main()` method for that particular exercise task, the compiler loses his mind and does not want to work with multiple `main()` methods...

ex2-4.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 // A 3D Vector struct
6 struct Vector
7 {
8     int a;
9     int b;
10    int c;
11 };
12
13 // Helper method to print a vector
14 void printVec(struct Vector *a)
15 {
16     printf("(%d, %d, %d), ", a->a, a->b, a->c);
17 }
18
19 // Helper method to print the Vector Array
20 void printArray(struct Vector *a, int size)
21 {
22     printf("[");
23     for (int i = 0; i < size; i++) {
24         printVec(a+i);
25     }
26     printf("]\n");
27 }
28
29 // Calculate L2 form of a vector
30 double absLength(const struct Vector *a)
31 {
32     return sqrt(a->a * a->a + a->b * a->b + a->c * a->c);
33 }
```

For the `qsort()` method to work, we need to provide a way to compare our `Vectors`. Apparently as parameters, "raw?" pointers should be used and only inside the function we cast them to be `Vector` pointers.

We then compare the two `Vectors` by their L2 form to decide if the first one is "greater", "even", or "lesser" than the second one.

ex2-4.c

```
35 // Compare Vectors, interesting parameter declaration to work with qsort()
36 int compareVec(const void *p, const void *q)
37 {
38     const struct Vector *a = (const struct Vector *)p;
39     const struct Vector *b = (const struct Vector *)q;
40
41     double lenA = absLength(a);
42     double lenB = absLength(b);
43
44     if (lenA > lenB) return 1;
45     if (lenA < lenB) return -1;
46     return 0;
47 }
```

Lastly we write a method to print our array of `Vectors` to a file. We open the file using the `fopen()` method, check if it really opened, then write to it using `fprintf()`. When we're finished, we `close` the file.

ex2-4.c

```
49 // write our vectors in a txt.
50 void writeToFile(struct Vector *a, int size)
51 {
52     FILE *file = fopen("vectors.txt", "w");
53
54     if (!file) // check if file is there
55     {
56         printf("Failed to open file");
57     }
58
59     // iterate through the array and write the vectors in the file.
60     for (int i = 0; i < size; i++)
61     {
62         fprintf(file, "(%d, %d, %d)\n", (a+i)->a, (a+i)->b, (a+i)->c);
63     }
64
65     fclose(file);
66 }
```

In our `main()` method we initialize our array using a for loop, then print it out, sort it with our `compareVec()` method and print it out again. At the end, we write our result to the file using our `writeToFile()` method.

ex2-4.c

```
68 int main() {
69
70     // Fill the array with some vectors.
71     struct Vector vectorArray[12];
72     struct Vector *arr_ptr = &vectorArray[0];
73
74     for (int i = 0; i < 12; i++)
75     {
76         vectorArray[i] = (struct Vector){30-i, i, 2*i};
77     }
78
79     // Print the array before and after the qsort().
80     printArray(arr_ptr, 12);
81     qsort(vectorArray, 12, sizeof(struct Vector), compareVec);
82     printArray(arr_ptr, 12);
83
84     writeToFile(arr_ptr, 12);
85 }
```