Introduction

This assignment is a C programming project where you write a program that *simulates* the organization of the virtual memory in a running process. In practice, you will create a dynamic *doubly linked* list for storing integers. The integers will be stored and removed into two *stacks*, which will grow and shrink during the program execution. In particular,

- the first stack will contain *odd positive integers*, e.g. 1, 3, or 9, start at the *head* of the list, and grow towards the tail,
- the second stack will contain *even nonnegative integers*, e.g. 0, 2, or 8, start at the *tail* of the list, and grow towards the head,
- between the two lists, the program will maintain a *free gap*, i.e. a set of free nodes, simulating the free memory gap between the heap and the stack in a process virtual address space.

Integers will be capped at some given value, `MAX`, fixed at the beginning of the program and the user will control the number of integers pushed to and pulled from the list from the terminal. Here is an illustrative diagram of the list that your program is supposed to build:

head	odd integers	$\rightarrow$ free space $\leftarrow$	even integers	tail

The list will grow or shrink by adding, when needed, given amounts of free nodes to the free gap between the stacks. The number of free nodes added to the list will be the same at each time and fixed by a macro, `BLOCKSIZE`, defined at the beginning of the program. As all integers stored in the list will be nonnegative, free nodes will be marked with the *fixed negative value* -1. To keep track of all dynamic memory allocations, you will use a *leak-aware* allocator and deallocator that update a shared counter defined in `main`.

You will write your program in four main steps, described in the following sections. For each section, save a standalone code that can be compiled without errors and produces the expected output. We suggest you call your solutions `step1.c`, `step2.c`, `step3.c`, `step4.c` and test the functions that you implement in each section through this Moodle Checker. As we also evaluate the structure and style of your programs, passing all tests will not guarantee you to get full marks.

Work and save your solutions on the teaching server to avoid unpleasant surprises, instabilities, and to be sure you can reproduce the examples given in this document without possible machine dependent issues. Always compile your code with

```
gcc —Wall —Werror —Wall
```

and check that the program executes without issues by running the executable, `a.out`, with the following command

```
valgrind ./a.out
```

and reading carefully what is printed on the screen.

Sketch of the implementation.

Step 1 (20 marks). In the beginning, the list will consist of only two nodes, the *head* and the *tail*, connected to each other. Your first task is to allocate these two nodes and *initialise* the doubly linked list.

Step 2 (20 marks). Before you start storing integers into the list, you will need to allocate a certain amount of *free nodes* and place them between the head and the tail. All allocated nodes, i.e. the head, the free nodes, and the tail, should *always* form a doubly-linked list.

Step 3 (40 marks). To store a given integer into the list, you will write a *push* function that adds the integer to the head-side stack, i.e. on the left of the free space, or the tail-side stack, i.e. on the right of the free space, depending on whether the integer is odd or even. New integers will be stored into the first available free node, i.e. odd integers into the leftmost node of the free gap and even integers into the rightmost node of the free gap. To remove an integer from the list, you will write a *pull* function that replaces a stored integer with -1, i.e. that makes the corresponding node free to store a new value. The choice between removing an integer from the left or the right will be fixed by a parameter passed to the pull function. To simplify the setup, you will always remove the *last added* integer of a stack, i.e. the last integer on the right of the head-side stack or the last integer on the left of the tail-side stack. The number of free nodes in the gap between the two stacks will then increase or decrease, according to the number of times you call the push and pull functions. You will need to keep track of the amount of available space, i.e. the number of free nodes in the gap, and

- remove, i.e. deallocate, `BLOCKSIZE` free nodes if the size of the gap is larger than `BLOCKSIZE`
- add, i.e. allocate, `BLOCKSIZE` free nodes if the size of the free gap is 0.

The allocation or deallocation process will simulate the request for more memory that a program can send to the operating system during the execution. Possible calls to the block allocation and deallocation should be included in the push and pull functions.

Step 4 (20 marks). In the end, you will make the program interactive so that the user can decide the number of integers to push to the list and the number of integers to pull from it. In practice, the user will enter a line of (noisy) signed integers into the terminal, e.g.

```
—one plus2 +two3 —4 +five5     —1 zero
```

and the program will perform the corresponding series of pulling and pushing operations, e.g.

1. add three integers
2. attempt to remove four integers
3. add five integers
4. attempt to remove 1.

After processing the user input, the program should clear the list, i.e. remove all integers, and deallocate all the allocated dynamic memory. The execution of the user instruction above will produce an output similar to

```
pushing 3 integers
 | 0 |  | 1 |  | 2 |  | 0 |  | 0 |
pulling 4 integers
 | 0 |  | 0 |
pushing 5 integers
 | 0 |  | 1 |  | 3 |  | −1 |  | 4 |  | 2 |  | 0 |  | 0 |
pulling 1 integers
 | 0 |  | 1 |  | 3 |  | −1 |  | −1 |  | 2 |  | 0 |  | 0 |
```

(Step 4, 20 marks).

1 Step 1: initialise the list (20 marks)

In this section, you will define the structures that you need to handle the nodes of the list and the list as a unit and write three functions to allocate, initialise, and deallocate the list starting nodes, i.e. the list head and tail. In particular,

- `allocator` will allocate a single node with `malloc` and update an integer counter

- **deAllocator** will free a single node with **free** and update the same counter
- **initialiseList** will allocate two nodes, the list head and tail, by calling **allocator** and initialise them.

Start by defining a node as

```
struct node {
  int i;
  struct node *next;
  struct node *prev;
};
```

where **i** is the integer stored in the node, **next** a pointer to the next node, i.e. the right neighbour of the node, and **prev** a pointer to the previous node, i.e. the left neighbour of the node. To make your program more readable, and let functions modify the list directly, define the following list handle

```
struct list {
  struct node *head;
  struct node *tail;
  struct node *right;
  struct node *left;
  int length;
};
```

where **head** is a pointer to the head of the list, i.e. the leftmost node of the list, **tail** is a pointer to the tail of the list, i.e. the rightmost node in the list, **right** is a pointer to the node storing the last-added even integer, and **left** is a pointer to the node storing the last-added odd integer. The following program allocates the list head and tail, connects them to each other, sets the structure members **left** and **right** to point to the head and tail of the list, initialises the size of the free-gap to 0, prints the list and the value of the allocation counter, frees the two nodes, and prints the counter again.

```
int main() {
  int counter = 0;
  struct list myList;
  printf("allocating two nodes  ... \n");
  myList.head = malloc(sizeof(struct node));
  counter++;
  myList.tail = malloc(sizeof(struct node));
  counter++;
  printf("initialising the list ... \n");
  myList.left = myList.head;
  myList.right = myList.tail;
  myList.tail->i = 0;
  myList.head->i = 0;
  myList.head->next = myList.tail;
  myList.head->prev = NULL;
  myList.tail->prev = myList.head;
  myList.tail->next = NULL;
  printf(" | %d || %d |\n", myList.head->i, myList.tail->i);
  printf("counter = %d\n", counter);
  printf("freeing the list ... \n");
  free(myList.head);
  counter--;
  free(myList.tail);
  counter--;
  printf("counter = %d\n", counter);
}
```

Code 1: Explicit initialisation

To compile the program, you will need to add all required headers, macros, structure definitions, and function declarations before `main` and all required function definitions after it. Your task consists of writing three functions declared as

```
void *allocator(int size, int *counter);
void deAllocator(void *p, int *counter);
void initialiseList(struct list *pList, int *counter);
```

where the return value of `allocator` is the pointer returned by `malloc`, `size` is the number of bytes that you want to allocate using `malloc`, `counter` is a pointer to the counter that keeps track of the number of dynamic memory allocations, `p` is a pointer to the memory block that you want to free using `free`, and `pList` is the pointer to the list-handle defined above. The functions should be implemented so that the following program becomes *equivalent* to Code 1, e.g. the output of the two programs should match for any initialisation of the counter.

```
int main() {
        int counter = 0;
        struct list myList;
        printf("allocating two nodes ... \n");
        printf("initialising the list ... \n");
        initialiseList(&myList, &counter);
        printf(" | %d || %d |\n", myList.head->i, myList.tail->i);
        printf("counter = %d\n", counter);
        printf("freeing the list ... \n");
        deAllocator(myList.head, &counter);
        deAllocator(myList.tail, &counter);
        printf("counter = %d\n", counter);
}
```

Code 2: Compact initialisation

In particular,

- `allocator` should
  - allocate a block of memory of `size` bytes by calling `malloc`
  - check if `malloc` returned a valid, i.e. non-null, pointer and, if so, increase the value of the counter by one
  - return the pointer returned by `malloc`

  (see Lines 5-6 of Code 1)

- `deAllocator` should
  - check that the first argument is a valid, i.e. non-null, pointer and, if so, free the memory block pointed by the first argument and decrease the counter by one
  - return nothing

  (see Lines 21-22 of Code 1)

- `initialiseList` should
  - call `allocator` twice, once to allocate the head and once to allocate the tail of the list
  - initialise the member of the structure as in Lines 10-17 of Code 1
  - return nothing.

Create a C file, `step1.c`, containing all required headers, macros, and function declarations, `main` given in Code 2, and your implementation of all functions. Check that the file can be compiled and run without errors and compare its output with the output of Code 1.

Example A run of `step1.c` should produce the following output

```
allocating two nodes  ...
initialising the list ...
 | 0 || 0 |
counter = 2
freeing the list ...
counter = 0
```

2 Step 2: allocate free nodes (20 marks)

In this section, you will make the program allocate extra free space. Free nodes will be added to the list in blocks of `BLOCKSIZE` nodes, where `BLOCKSIZE` is a macro defined as

```
#define BLOCKSIZE 5
```

Your task is to write a function declared as

```
void allocateBlock(struct list *pList, int *counter)
```

that adds `BLOCKSIZE` nodes on the right of the node pointed by `pList->left`. The function should consist of a loop of `BLOCKSIZE` iterations where, at each iteration, you

- allocate a new object of type `struct node` with `allocator`
- link the new node to the existing ones so that the doubly-linked structure of the list is preserved
- set `i` of the new node to -1
- increase `pList->length` by one

To preserve the doubly-linked structure of the list at every step, you need to

- let `prev` of the new node be a pointer to the node pointed by `pList->left`
- let `next` of the new node be a pointer to the node pointed by `pList->left->next`
- let `prev` of the node pointed by `pList->left->next` be a pointer to the new node
- let `next` of the node pointed by `pList->left` be a pointer to the new node

To get an intuition of what the function is supposed to do, draw a simple cartoon of the node-insertion process. Copy `step1.c` into a new file, call it `step2.c`, and replace `main` with

```
int main() {
  int counter = 0;
  struct list myList;
  printf("initialising the list ... \n");
  initialiseList(&myList, &counter);
  printList(&myList, &counter);
  printf("allocating %d free nodes ... \n", BLOCKSIZE);
  allocateBlock(&myList, &counter);
  printList(&myList, &counter);

  printf("freeing %d free nodes ... \n", BLOCKSIZE);
  deAllocateBlock(&myList, &counter);
  printList(&myList, &counter);
  printf("freeing head and tail ... \n");
  removeList(&myList, &counter);
  printList(&myList, &counter);
}
```

Code 3: Free node allocation

Check that the file contains all required headers, macros, structure definitions and function declarations before `main` and all functions definitions after it. The code of `deAllocateBlock`, `printList`, and `removeList` are given in Appendix A.

Example If you set `BLOCKSIZE` to 5, the output should be

```
initialising the list ...
 | 0 |  | 0 |
counter = 2
allocating 5 free nodes ...
 | 0 |  | -1 |  | -1 |  | -1 |  | -1 |  | -1 |  | 0 |
counter = 7
freeing 5 free nodes ...
 | 0 |  | 0 |
counter = 2
freeing head and tail ...
counter = 0
```

3 Step 3: push and pull integers to the list (40 marks)

In this section, you will write two functions, `pushInt` and `pullInt`, that store and remove integers from the list. The push function will need to check that the list has a *nonempty* free gap, i.e. if `myList.length` is not zero, before attempting to store a new integer into it. If the free gap is empty, the push function should call `allocateBlock` to increase the size of the list. After removing an integer, the pull function will check that the size of the free gap is not *too big*, i.e. larger than `BLOCKSIZE`, and, in that case, will call `deAllocateBlock` to reduce the size of the list.

`pushInt`: Start by writing `pushInt`, the function that pushes a given integer to the list. The function should be declared as

```
void pushInt(struct list *pList, int *counter, int i)
```

where `pList` is a pointer to the list handle defined above, e.g. `myList` in Code 2, and `i` is an integer. In particular, `pushInt` should

- call `allocateBlock` if there are no free nodes in the gap, i.e. if `pList->length` is 0, then
- store a new *odd* integer on the first available node on the right of head-side stack, i.e. on the node on the right of the node pointed by `pList->left`, or
- store a new *even* integer on the first available node on the left of the tail-side stack, i.e. on the node on the left of the node pointed by `pList->right`.

To test your implementation, call `pushInt` from Line 10 of Code 3 with some arbitrary integer as the third parameter.

`pullInt`: If everything looks good, proceed and write `pullInt`, the function that pulls an integer from the list. The function should be declared as

```
void pullInt(struct list *pList, int *counter, int i)
```

where `pList` is a pointer to the list handle defined above and `i` an integer. In this case, you will only use `i` to check if it is even or odd and decide on what side of the list you will pull the integer, i.e. to decide whether you will remove the odd integer stored in the node pointed by `pList->left`, or the even integer in the node pointed by `pList->right`. We suggest you start from the code of `pushInt` and tweak it so that `pullInt`

- accepts the same parameters as `pushInt` and returns nothing

- if `i` is odd and $\texttt{pList}->\texttt{left} \neq \texttt{pList}->\texttt{head}$, i.e. there is at least one odd integer in the list,
  - replaces the integer stored in the node pointed by `pList->left` with -1
  - moves `pList->left` to `pList-left->prev`
  - increases `pList->length` by one
- if `i` is even and $\texttt{pList}->\texttt{right} \neq \texttt{pList}->\texttt{tail}$, i.e. there is at least one even integer in the list,
  - replaces the integer stored in the node pointed by `pList->right` with -1
  - moves `pList->right` to `pList-right->next`
  - increases `pList->length` by one
- checks, independently on whether `pList->length` has been increased or not, if there are too many free nodes, i.e. if $\texttt{pList}->\texttt{length} \geq \texttt{BLOCKSIZE}$ and, if so, calls `deAllocateBlock` to remove `BLOCKSIZE` free nodes from the list

Verify that your implementation is correct by calling `pullInt` to remove the integer that you store in the list with `pushInt`. Print the list with `printList` before and after the pull operation to check if you see what you expect. Copy `step2.c` into a new file, `step3.c` and add the declaration of `pushInt` and `pullInt` to the function declaration list and their implementation on the bottom of the file. In `main` of `step3.c`, replace Lines 7-13 of Code 3 with

```
int N = 4 ;
int i = 0;
while (i < N) {
  pushInt(&myList, &counter, i * i);
  printList(&myList, &counter);
  i++;
  }
while (i > 1) {
  pullInt(&myList, &counter, i * i + 1);
  printList(&myList, &counter);
  i--;
  }
while (myList.length != 0) {
  pullInt(&myList, &counter, 1);
  pullInt(&myList, &counter, 2);
  }
```

save, compile and run the program with Valgrind to check that everything is correct.

Example. If you set `BLOCKSIZE` to 3, the output should be

```
initialising the list ...
 | 0 |  | 0 |
counter = 2
 | 0 |  | -1 |  | -1 |  | 0 |  | 0 |
counter = 5
 | 0 |  | 1 |  | -1 |  | 0 |  | 0 |
counter = 5
 | 0 |  | 1 |  | 4 |  | 0 |  | 0 |
counter = 5
 | 0 |  | 1 |  | 9 |  | -1 |  | -1 |  | 4 |  | 0 |  | 0 |
counter = 8
 | 0 |  | 1 |  | 4 |  | 0 |  | 0 |
counter = 5
 | 0 |  | 1 |  | -1 |  | 0 |  | 0 |
counter = 5
 | 0 |  | -1 |  | -1 |  | 0 |  | 0 |
counter = 5
freeing head and tail ...
counter = 0
```

Try different values of `BLOCKSIZE` to see the changes.

4 Step 4: interact with the user (20 marks)

In this section, you will make your program interactive and write a function, `getSignedInt`, that you can call to parse an input string of words and translate it into signed integers. Positive and negative integers will determine the number of `pushInt` and `pullInt` calls made by the program. For example, `-one12two` will make the program pull 12 integers from the list.

`getSignedInt.` Write a function that translates a word typed by the user in the terminal into the corresponding *signed* integer. The function should be declared as

```
int getSignedInt(int *n)
```

where `n` is a pointer to the integer variable that will store the obtained integer and the return value should be

- 1 if the last parsed character is *not* a newline character
- 0 otherwise

If the first processed character is an empty space, the function should return 1 immediately and without doing anything. This is to ensure that the program behaves correctly even if the user enters some unexpected extra spaces between words. The input should be processed as suggested in Algorithm 1, where `MAX` is a macro defined on the top of your file as

```
#defined MAX 10
```

For example,

```
+3 -one1 plus2 -1 +45
```

will produce four valid signed integers, $+3$, -1, -1, and $+5$, if you set `MAX` to 10 as suggested above, and $+3$, -1, -1, and $+3$ if you set `MAX` to 6 as $3 = 45\%6$.

`main` Finally, complete `main` given in Code 4 so that it behaves as in the following example

```
initialising the list ...
 | 0 |  | 0 |
counter = 2
-one plus2 +two3 -4 +five5    -1 zero
pushing 3 integers
 | 0 |  | 1 |  | 2 |  | 0 |  | 0 |
counter = 5
pulling 4 integers
 | 0 |  | 0 |
counter = 2
pushing 5 integers
 | 0 |  | 1 |  | 3 |  | -1 |  | 4 |  | 2 |  | 0 |  | 0 |
counter = 8
pulling 1 integers
 | 0 |  | 1 |  | 3 |  | -1 |  | -1 |  | 2 |  | 0 |  | 0 |
counter = 8
removing all integers ...
 | 0 |  | 0 |
counter = 2
freeing head and tail ...
counter = 0
```

Algorithm 1 Pseudocode of `int getSignedInt(int *n)`

```
Require: int *n, MAX
  set *n to 0
  declare a char variable, c
  read a single character from the terminal and store the character into c
  if c is an empty space then
      return 1
  end if
  if c is a newline character then
      return 0
  end if
  declare an int variable, flip
  if c = $'-'$ then
      set flip to $-1$
  else if c = $'+'$ then
      set flip to $+1$
  else
      set flip to 0
  end if
  while c is not an empty space or a newline character do
      read a single character from the terminal
      if c is a numerical character then
          let $*n = *n * 10$
          let $*n = *n + x$, where $x$ is the integer value represented by character c
      end if
  end while
  cap *n to MAX by letting $*n = *n \% MAX$
  make the integer signed by letting $*n = flip * *n$
  if c is a newline character then
      return 0
  else
      return 1
  end if
```

```c
... //all required headers
... // all required macros
... //all required structure definitions
... //all required function declarations
int main() {
  int counter = 0,  n = 0, go = ...;
  struct list myList;
  printf("initialising the list ... \n");
  initialiseList(&myList, &counter);
  printList(&myList, &counter);
  while (...) {
    go = ...;
    if (...) printf("pushing %d integers\n", abs(n));
    if (...) printf("pulling %d integers\n", abs(n));
    int j = 0;
    while (j < abs(n)) {
      if (...) pushInt(&myList, &counter, j);
      if (...) pullInt(&myList, &counter, j);
      j++;
    }
    if (...) printList(&myList, &counter);
  }
  if ((myList.head != myList.left) || (myList.tail != myList.right)) {
    printf("removing all integers ... \n");
  while ((myList.head != myList.left) || (myList.tail != myList.right)) {
    pullInt(&myList, &counter, ...);
    pullInt(&myList, &counter, ...);
  }
  printList(&myList, &counter);
  printf("freeing head and tail ... \n");
  removeList(&myList, &counter);
  printList(&myList, &counter);
}
... //all required function definitions
```

Code 4: `main`

Save the program into a file called `step4.c`, compile it with `gcc -Wall -Werror -Wpedantic` and run it with Valgrind to see if it executes without errors.

Example. If you set both `BLOCKSIZE` and `MAX` to 3 and enter the following input

```
one −2 +three3 −45fortyfive 6 minus7 +8   −9
```

the output should be

```
./a.out
initialising the list ...
 | 0 |  | 0 |
counter = 2
one −2 +three3 −45fortyfive 6 minus7 +8   −9
pulling 2 integers
 | 0 |  | 0 |
counter = 2
pushing 2 integers
 | 0 |  | 1 |  | −1 |  | 0 |  | 0 |
counter = 5
removing all integers ...
 | 0 |  | 0 |
counter = 2
freeing head and tail ...
```

```
counter = 0
```

The same `stdin` input, but when you set both `BLOCKSIZE` and `MAX` to 5, produces

```
./a.out
initialising the list ...
 | 0 |  | 0 |
counter = 2
one -2 +three3 -45fortyfive 6 minus7 +8   -9
pulling 2 integers
 | 0 |  | 0 |
counter = 2
pushing 3 integers
 | 0 |  | 1 |  | -1 |  | -1 |  | 2 |  | 0 |  | 0 |
counter = 7
pushing 3 integers
 | 0 |  | 1 |  | 1 |  | -1 |  | -1 |  | -1 |  | -1 |  | 2 |  | 0 |  | 2 |  | 0 |  | 0
    |
counter = 12
pulling 4 integers
 | 0 |  | -1 |  | -1 |  | -1 |  | 2 |  | 0 |  | 0 |
counter = 7
removing all integers ...
 | 0 |  | 0 |
counter = 2
freeing head and tail ...
counter = 0
```

A Auxiliary functions

Implementation of some of the auxiliary functions called in Code 3

```c
void printList(struct list *pList, int *counter) {
  if (pList->head) {
    struct node *cur = pList->head;
    while (cur){
      printf(" | %d | ", cur->i);
      cur = cur->next;
    }
    printf("\n");
  }
  printf("counter = %d\n", *counter);
}
```

Code 5: `printList`

```c
void removeList(struct list *pList, int *counter) {
  deAllocator(pList->head, counter);
  deAllocator(pList->tail, counter);
  pList->head = NULL;
  pList->tail = NULL;
  pList->length = 0;
  pList->left = NULL;
  pList->right = NULL;
}
```

Code 6: `removeList`

```
void deAllocateBlock(struct list *pList, int *counter) {
  int i = 0;
  while (i < BLOCKSIZE) {
    struct node *temp = pList->left->next;
    pList->left->next->next->prev = pList->left;
    pList->left->next = pList->left->next->next;
    deAllocator(temp, counter);
    pList->length = pList->length -1;
    i++;
  }
}
```

Code 7: `deAllocateBlock`