202223 CS2850 CW2: C mini-project

Opened: Wednesday, 7 December 2022
Close: Monday, 9 January 2023, at 10 am

Submit this assignment using the Moodle submission link 202223 CS2850 CW2: C mini-projectby Monday, 9 January 2023, at 10 am. Feedback will be provided in the coursework grid of the course Moodle page in approximately 10 working days.

Learning outcomes

To complete this assignment you should know the high-level structure of the address space in a running process and how to write a C program where you

- declare, define, and use, variables and functions,
- parse `stdin` input,
- allocate and free the memory dynamically, and
- use a doubly-linked list to store data.

Marking criteria

Submissions are assessed on functionality and coding style. Try to write readable, well-formatted and well-commented code. More importantly, be sure that all your programs can be compiled using `gcc -Wall -Werror -Wpedantic` and run without errors on `linux.cim.rhul.ac.uk`. To spot possible execution issues, run the programs with Valgrind and check that the end of the output is

```
... == ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Before submitting, test your code with the assignment Moodle Code Checker. Passing all tests, however, will not guarantee you get full marks.

Instructions

The submission system allows you to submit a single archive file, which should contain all your programs. Create a compressed directory, e.g. `202223cs2850cw2.zip`, that contains the four C files described in the following sections , `step1.c`, `step2.c`, `step3.c`, and `step4.c`. Optionally, you can save all auxiliary functions in a (single) separate file, `functions.c`, that you include in a task-specific program, e.g. `step1.c`, by writing

```
#include "functions.c"
```

on top of the file. In this case, `step1.c` would have the following structure

```
//... macro definitions
#include "functions.c"
int main() {
        ...
}
```

Be sure that **your name or ID does not appear anywhere** in your submission. Your files will be recompiled and run in the submission directory using the teaching server compiler. See Section 5 for more information on submission rules and extensions.

Academic misconduct

Coursework submissions are routinely checked for academic misconduct (working together, copying from sources, etc.). Penalties can range from a 10% deduction of the assignment mark, zero for the assignment or the case being referred to a Senior Vice-Principal to make a decision (for repeat offences). Further details can be found here.

Overview

In this assignment, you will write a program that *simulates* the organization of the virtual memory in a running process. The virtual address space will be represented by a dynamic *doubly linked* list. The list will store integers in two *stacks*, which will grow and shrink during the program execution. In particular,

- the first stack, which starts at the *head* of the list and grow towards the tail, will contain *odd* digits, i.e. 1, 3, 5, 7, or 9,
- the second stack, which starts at the *tail* of the list and grows towards the head, will contain *even* digits, e.g. 0, 2, 4, 6, or 8,
- between the two stacks, the program will maintain a *free space*, i.e. a set of free nodes, simulating the free memory between the heap and the stack in a process virtual address space.

Here is an illustrative diagram of the list that your program is supposed to build:

head	odd integers	→	free space	←	even integers	tail

The list will grow or shrink by adding or removing fixed-size blocks of *free nodes*. The number of free nodes added to or removed from the list will be fixed by a macro, `BLOCKSIZE`, defined at the beginning of the program. Newly allocated free nodes will temporarily store a fixed negative value, e.g. -1. When some free space is available, the program will be able to *push* digits to the list. The program will also include a *pull* function, to free one node from a selected stack. The user will control the pushing and pulling operations by writing *two strings* in the terminal. The number of digits accepted from the input will be capped at a fixed value to avoid overflow. The program will keep track of all dynamic memory allocations through *leak-aware* allocation and deallocation functions, which update a counter defined in `main`.

To write the program, follow the steps described in the next sections. Save a separate C file, e.g. `step1.c`, `step2.c`, `step3.c`, or `step4.c`, for each section and make sure that it can be compiled without errors and produces the expected output. After completing a section, test your implementation with Valgrind and the Moodle Code Checker.

Example. The output of the final program with `BLOCKSIZE` set to 1 and input `0n3 plu8 tw0 3qu4l th833` and `000` is

```
 | 0 |  | 0 |
counter = 2
0n3 plu8 tw0 3qu4l th833
000
 | 0 |  | 0 |  | 0 |
counter = 3
 | 0 |  | -1 |  | 0 |
counter = 3
 | 0 |  | 3 |  | 0 |
counter = 3
 | 0 |  | 3 |  | 0 |
counter = 3
 | 0 |  | 3 |  | 8 |  | 0 |
counter = 4
 | 0 |  | 3 |  | -1 |  | 0 |
counter = 4
 | 0 |  | 3 |  | 0 |  | 0 |
counter = 4
 | 0 |  | 3 |  | 3 |  | 0 |  | 0 |
counter = 5
 | 0 |  | 3 |  | 3 |  | 0 |  | 0 |
counter = 5
```

```
 ||
counter = 0
```

The same input, with `BLOCKSIZE` set to 2, produces

```
 | 0 |  | 0 |
counter = 2
0n3 plu8 tw0 3qu4l th833
000
 | 0 |  | -1 |  | 0 |  | 0 |
counter = 4
 | 0 |  | -1 |  | -1 |  | 0 |
counter = 4
 | 0 |  | 3 |  | -1 |  | 0 |
counter = 4
 | 0 |  | 3 |  | -1 |  | 0 |
counter = 4
 | 0 |  | 3 |  | 8 |  | 0 |
counter = 4
 | 0 |  | 3 |  | -1 |  | 0 |
counter = 4
 | 0 |  | 3 |  | 0 |  | 0 |
counter = 4
 | 0 |  | 3 |  | 3 |  | -1 |  | 0 |  | 0 |
counter = 6
 | 0 |  | 3 |  | 3 |  | 4 |  | 0 |  | 0 |
counter = 6
 | 0 |  | 3 |  | 3 |  | -1 |  | 8 |  | 4 |  | 0 |  | 0 |
counter = 8
 | 0 |  | 3 |  | 3 |  | 3 |  | 8 |  | 4 |  | 0 |  | 0 |
counter = 8
 | 0 |  | 3 |  | 3 |  | 3 |  | 3 |  | -1 |  | 8 |  | 4 |  | 0 |  | 0 |
counter = 10
 | 0 |  | 3 |  | 3 |  | 3 |  | 3 |  | -1 |  | 8 |  | 4 |  | 0 |  | 0 |
counter = 10
 ||
counter = 0
```

1 Step 1: initialise the list (20 marks)

In this section, you will define the structures that represent the nodes of the list and the list as a unit. You will also write four auxiliary functions: `allocator`, which allocates a single node with `malloc` and updates an integer counter, `deAllocator`, which frees a single node with `free` and updates the same counter, `initialiseList`, which calls `allocator` to allocate the list head and tail and initialise them, and `freeList`, which calls `deAllocator` to free the head and the tail of the list and set all pointers to `NULL`. 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` is a pointer to the next node, i.e. the right neighbour of the node, and `prev` is a pointer to the previous node, i.e. the left neighbour of the node. The second structure you need is a *list handle*, containing all information you need to access the list, e.g.

```
struct list {
  struct node *head;
```

```
  struct node *tail;
  struct node *right;
  struct node *left;
  int length;
};
```

where `head` is a pointer to the list head, i.e. the leftmost node of the list, `tail` is a pointer to the list tail, i.e. the rightmost node in the list, `left` is a pointer to the node storing the last-added odd integer, `right` is a pointer to the node storing the last-added even integer, and `length` is the number of nodes in the free space. The program in Listing 1 declares a list called `myList`, allocates the head and tail of the list and connects them, makes `myList.left` and `myList.right` point to the head and tail of the list, initialises `myList.length` to 0, prints the list and the value of the allocation counter, frees the nodes, set all list pointers to `NULL` and `length` to -1, and prints the counter again.

```
int main() {
  int counter = 0;
  struct list myList;
  myList.head = malloc(sizeof(struct node));
  counter++;
  myList.tail = malloc(sizeof(struct node));
  counter++;
  myList.head->next = myList.tail;
  myList.tail->prev = myList.head;
  myList.head->prev = NULL;
  myList.tail->next = NULL;
  myList.head->i = 0;
  myList.tail->i = 0;
  myList.left = myList.head;
  myList.right = myList.tail;
  myList.length = 0;
  printList(&myList, &counter);
  if (!myList.length && myList.head == myList.left && myList.tail == myList.right) {
    free(myList.head);
    counter--;
    free(myList.tail);
    counter--;
    if (!counter) {
      myList.head = NULL;
      myList.left = NULL;
      myList.tail = NULL;
      myList.right = NULL;
      myList.length = -1;
    }
  }
  printList(&myList, &counter);
}
```

Listing 1: Explicit initialisation

The auxiliary function `printList` is defined in Appendix A. After adding all required definitions, compile the code and run it. The output should be

```
 | 0 |  | 0 |
counter = 2
 ||
counter = 0
```

The program in Listing 1 is not very readable. Rewrite it as

```
int main() {
  int counter = 0;
```

```
  struct list myList;
  initialiseList(&myList, &counter);
  printList(&myList, &counter);
  freeList(&myList, &counter);
  printList(&myList, &counter);
}
```

Listing 2: Compact initialisation

by defining the following auxiliary functions:

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

Define the functions so that the programs in Listing 1 and Listing 2 have the same output. In particular,

- `allocator` should
  - allocate `size` bytes of memory by calling `malloc`,
  - check if `malloc` returned a valid pointer, i.e. a non-null pointer, and, if so, increase the value of the counter by one, and
  - return the pointer returned by `malloc`.
- `deAllocator` should
  - check that the first argument is a valid, i.e. non-null, pointer and, if so, free the memory pointed by the first argument and decrease the counter by one and
  - return nothing.
- `initialiseList` should
  - call `allocator` to allocate the head and the tail of the list,
  - initialise the members of the list as in Listing 1, and
  - return nothing.
- `freeList` should
  - check that `pList->left` points to the same node as `pList->head`, `pList->right` points to the same node as `pList->tail`, and `pList->length` is zero, i.e. that the list does not contain any occupied or free nodes,
  - if the list is empty, call `deAllocator` twice to deallocate the head and the tail of the list, and
  - if the deallocation is successful, set all pointers to `NULL` and `length` to -1.

Create a C file, `step1.c`, containing all required headers and definitions and the code in Listing 2. Compile the code and check that it behaves *exactly* as the program in Listing 1.

2 Step 2: allocate free nodes (20 marks)

In this section, you will make the program allocate extra free space in blocks of size `BLOCKSIZE`. To make the block size visible to all functions, define a macro called `BLOCKSIZE` just after the headers. The main task of this step is to define a free-block allocator, declared as

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

where `pList` is a pointer to the list handle, `counter` is a pointer to the memory allocation counter defined in `main`, and `nNodes` is the number of nodes to be added to the free space. To allocate new nodes, you will call `allocateBlock` with `BLOCKSIZE` as the third argument.

- Let `allocateBlock` consist of a loop of `nNodes` iterations where, at each iteration, you
  - allocate a new object of type `struct node` by calling `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, and
  - increase `pList->length` by one.

To get an intuition of what the function is supposed to do, draw a simple cartoon of the node-insertion process. Copy the content of `step1.c` into a new file, `step2.c`, add the definition of `BLOCKSIZE` on the top, replace `main` with

```
int main() {
  int counter = 0;
  struct list myList;
  initialiseList(&myList, &counter);
  printList(&myList, &counter);
  allocateBlock(&myList, &counter, BLOCKSIZE);
  printList(&myList, &counter);
  deAllocateBlock(&myList, &counter, BLOCKSIZE);
  printList(&myList, &counter);
  freeList(&myList, &counter);
  printList(&myList, &counter);
}
```

Listing 3: Free node allocation

and include your implementation of `allocateBlock` and the definition of `deAllocateBlock` given in Appendix A. Compile the code and check that the output is the same as in the following example.

Example. If you set `BLOCKSIZE` to 2, the output should be

```
 | 0 |  | 0 |
counter = 2
 | 0 |  | -1 |  | -1 |  | 0 |
counter = 4
 | 0 |  | 0 |
counter = 2
 ||
counter = 0
```

and setting `BLOCKSIZE` to 1 gives

```
 | 0 |  | 0 |
counter = 2
 | 0 |  | -1 |  | 0 |
counter = 3
 | 0 |  | 0 |
counter = 2
 ||
counter = 0
```

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

In this section, you will write three functions, `pushInt`, to store a digit in the list, `pullInt`, to remove a digit from the list, and `clearList`, which calls `pullInt` until the list is completely free, deallocate the free nodes, and call `freeList` to free the head and the tail of the list. `pushInt` will check that the list has a *nonempty* free space, i.e. if `length` is not zero, and, if so, store a new digit in it. If the free space is empty, `pushInt` will call `allocateBlock` to increase its size and *then* perform the required push operation. `pullInt` will

remove a digit from one side of the list and *then* check if the size of the free space is larger than `BLOCKSIZE`. In that case, `pullInt` should call `deAllocateBlock` to deallocate `BLOCKSIZE` free nodes.

- Declare `pushInt` as

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

  where `pList` and `counter` are interpreted as in the previous section and `i` is an integer from 0 to 9. `pushInt` should

  - call `allocateBlock`, with `BLOCKSIZE` as a third argument, if there are no free nodes in the gap,
  - check if `i` is odd or even,
  - if `i` is *odd*, store it on the first available node on the right of the head-side stack, i.e. on the node on the right of the node pointed by `pList->left`, and,
  - if `i` is *even*, store it 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`.

- Declare `pullInt` as

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

  where `pList`, `counter`, and `i` are interpreted as in `pushInt`. In this case, `i` is used to decide on what side of the list you perform the pull operation, i.e. to decide whether you remove the (odd) digit in the node pointed by `pList->left` or the (even) digit in the node pointed by `pList->right`. `pullInt` should

  - check if `i` is odd or even,
  - if `i` is *odd*, check that `pList->left` $\neq$ `pList->head`, i.e. there is at least one odd integer in the list, and, if so,
    - replace the digit in the node pointed by `pList->left` with -1,
    - move `pList->left` to `pList->left->prev`, and
    - increase `pList->length` by one,
  - if `i` is even, check that `pList->right` $\neq$ `pList->tail`, i.e. there is at least one even integer in the list, and, if so,
    - replace the digit in the node pointed by `pList->right` with -1,
    - move `pList->right` to `pList->right->next`, and
    - increase `pList->length` by one,
  - check if `pList->length` $>$ `BLOCKSIZE` and, if so, call `deAllocateBlock` to remove `BLOCKSIZE` free nodes from the free space.

- Declare `clearList` as

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

  where `pList` and `counter` are interpreted as in `pushInt` and `pullInt`. `clearList` should

  - remove all digits in the head-side stack by calling `pullInt` with `i` $= 1$ until `pList->left` points to the head of the list,
  - remove all digits in the tail-side stack by calling `pullInt` with `i` $= 0$ until `pList->right` points to the tail of the list,
  - remove all free nodes by calling `deAllocateBlock` with `pList->length` as the third argument, and
  - free the list by calling `freeList`.

Copy `step2.c` into a new file, `step3.c`, and replace `main` with

```
int main() {
  int counter = 0;
  struct list myList;
  initialiseList(&myList, &counter);
  printList(&myList, &counter);
  int N = 6;
  int i = 0;
  while (i < N) {
    pushInt(&myList, &counter, i % 9);
    printList(&myList, &counter);
    pullInt(&myList, &counter, 0);
    printList(&myList, &counter);
    i++;
  }
  clearList(&myList, &counter);
  printList(&myList, &counter);
}
```

Listing 4: Pushing and pulling

Compile the code and check that the output is the same as in the following example.

Examaple. If you set `BLOCKSIZE` to 2, the output should be

```
 | 0 |  | 0 |
counter = 2
 | 0 |  | -1 |  | 0 |  | 0 |
counter = 4
 | 0 |  | -1 |  | -1 |  | 0 |
counter = 4
 | 0 |  | 1 |  | -1 |  | 0 |
counter = 4
 | 0 |  | 1 |  | -1 |  | 0 |
counter = 4
 | 0 |  | 1 |  | 2 |  | 0 |
counter = 4
 | 0 |  | 1 |  | -1 |  | 0 |
counter = 4
 | 0 |  | 1 |  | 3 |  | 0 |
counter = 4
 | 0 |  | 1 |  | 3 |  | 0 |
counter = 4
 | 0 |  | 1 |  | 3 |  | -1 |  | 4 |  | 0 |
counter = 6
 | 0 |  | 1 |  | 3 |  | -1 |  | -1 |  | 0 |
counter = 6
 | 0 |  | 1 |  | 3 |  | 5 |  | -1 |  | 0 |
counter = 6
 | 0 |  | 1 |  | 3 |  | 5 |  | -1 |  | 0 |
counter = 6
 ||
counter = 0
```

and setting `BLOCKSIZE` to 1 gives

```
 | 0 |  | 0 |
counter = 2
 | 0 |  | 0 |  | 0 |
counter = 3
 | 0 |  | -1 |  | 0 |
counter = 3
```

```
 | 0 |  | 1 |  | 0 |
counter = 3
 | 0 |  | 1 |  | 0 |
counter = 3
 | 0 |  | 1 |  | 2 |  | 0 |
counter = 4
 | 0 |  | 1 |  | -1 |  | 0 |
counter = 4
 | 0 |  | 1 |  | 3 |  | 0 |
counter = 4
 | 0 |  | 1 |  | 3 |  | 0 |
counter = 4
 | 0 |  | 1 |  | 3 |  | 4 |  | 0 |
counter = 5
 | 0 |  | 1 |  | 3 |  | -1 |  | 0 |
counter = 5
 | 0 |  | 1 |  | 3 |  | 5 |  | 0 |
counter = 5
 | 0 |  | 1 |  | 3 |  | 5 |  | 0 |
counter = 5
 ||
counter = 0
```

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

To make your program interactive, you need a function, `getInput`, for parsing the user input. The program should ask the user to enter two (possibly noisy) lines of integers and parse them by calling `getInput` twice. The first line will contain the integers that the program will push to the list. The second line will contain the integers that decide from which side of the list the program will pull the integers. To avoid overflow, `getInput` will only accept `maxInput` digits, where `maxInput` $= 5 *$ `BLOCKSIZE`. For example, if `BLOCKSIZE` $= 1$, and the first line of the user input is

```
0n3 plu8 tw0 3qu4l th833
```

the program will push a 0 to the tail-side stack, a 3 to the head-side stack, an 8 and another 0 to the tail-side stack, and another 3 to the heas-side stack. All characters after the second 3 will be discarded. Define `maxInput` outside main, e.g. just after the headers, so that it can be accessed by all functions.

- Declare `getInput` as

  ```
  int getInput(char *s);
  ```

  where `s` is a string and the return value is the minimum between the number of digits in the input and `maxInput`. The function should use `s` to buffer the digits as suggested in Algorithm 1.

- Complete `main` given in Listing 5 so that your program behaves as in the examples given below.

  ```
  int main() {
    int counter = 0;
    struct list myList;
    initialiseList(&myList, &counter);
    printList(&myList, &counter);
    char sPush[...];
    char sPull[...];
    int lenPush = getInput(...);
    int lenPull = getInput(...);
    int i = 0;
    int j = 0;
    while ((i + j) < (lenPush + lenPull)) {
  ```

Algorithm 1 Pseudocode of `int getInput(char *s)`

Input: `char *s`
declare a variable, `c`, of type `char`
declare a variable, `i`, of type `int`
set `c` to the null-character
set `i` to 0
while `c` is not a new line character and `i` is smaller than `maxInput` **do**
    read a new character from the terminal and store it in `c`
    if `c` is a digit, i.e. `c` $\in \{0, \ldots, 9\}$ **then**
        copy `c` to the `i`th entry of `s`
        increase `i` by one
    end if
end while
if `i` is equal to `maxInput` **then**
    while `c` is not a new line character **do**
        read a new character from the terminal and do nothing
    end while
end if
return `i`

```
      if (i < ...) {                                          13
        pushInt(&myList, &counter, ...);                      14
        i++;                                                  15
        printList(&myList, &counter);                         16
      }                                                       17
      if (j < ...) {                                          18
        pullInt(&myList, &counter, ...);                      19
        j++;                                                  20
        printList(&myList, &counter);                         21
      }                                                       22
    }                                                         23
    printList(&myList, &counter);                             24
    clearList(&myList, &counter);                             25
    printList(&myList, &counter);                             26
  }                                                           27
```

Listing 5: `main`

Save the program in a new file, `step4.c`, and compile and run it for different choices of `BLOCKSIZE`. For any choice of `BLOCKSIZE` and different input strings, use Valgrind to see if the program runs without problems.

Note. The return value of `getInput` is the number of accepted digits contained in the user input. The function should copy the selected digits into `s` *without converting them to integers*. The conversion to integers should be performed in `main` when the digits are used as an input for `pushInt` and `pullInt`.

Example. If you set `BLOCKSIZE` to 1 and the user enters

```
one1 + two 02 =  three    33333333333                         1
```

and then

```
2 minus 1 minus 01 minus000001 = minus one, i.e. —1           1
```

the output should be

```
 | 0 |  | 0 |
counter = 2
```

```
one1 + two 02 =  three    33333333333
2 minus 1 minus 01 minus000001 = minus one, i.e. —1
 | 0 |  | 1 |  | 0 |
counter = 3
 | 0 |  | 1 |  | 0 |
counter = 3
 | 0 |  | 1 |  | 0 |  | 0 |
counter = 4
 | 0 |  | —1 |  | 0 |  | 0 |
counter = 4
 | 0 |  | 2 |  | 0 |  | 0 |
counter = 4
 | 0 |  | —1 |  | 0 |  | 0 |
counter = 4
 | 0 |  | 3 |  | 0 |  | 0 |
counter = 4
 | 0 |  | —1 |  | 0 |  | 0 |
counter = 4
 | 0 |  | 3 |  | 0 |  | 0 |
counter = 4
 | 0 |  | 3 |  | —1 |  | 0 |
counter = 4
 | 0 |  | 3 |  | —1 |  | 0 |
counter = 4
 ||
counter = 0
```

The same **stdin** input when you set both BLOCKSIZE to 2 produces

```
 | 0 |  | 0 |
counter = 2
one1 + two 02 =  three    33333333333
2 minus 1 minus 01 minus000001 = minus one, i.e. —1
 | 0 |  | 1 |  | —1 |  | 0 |
counter = 4
 | 0 |  | 1 |  | —1 |  | 0 |
counter = 4
 | 0 |  | 1 |  | 0 |  | 0 |
counter = 4
 | 0 |  | —1 |  | 0 |  | 0 |
counter = 4
 | 0 |  | 2 |  | 0 |  | 0 |
counter = 4
 | 0 |  | —1 |  | 0 |  | 0 |
counter = 4
 | 0 |  | 3 |  | 0 |  | 0 |
counter = 4
 | 0 |  | —1 |  | 0 |  | 0 |
counter = 4
 | 0 |  | 3 |  | 0 |  | 0 |
counter = 4
 | 0 |  | 3 |  | —1 |  | 0 |
counter = 4
 | 0 |  | 3 |  | 3 |  | 0 |
counter = 4
 | 0 |  | 3 |  | 3 |  | 0 |
counter = 4
 | 0 |  | 3 |  | 3 |  | 3 |  | —1 |  | 0 |
counter = 6
 | 0 |  | 3 |  | 3 |  | 3 |  | —1 |  | 0 |
```

```
counter = 6
 | 0 |  | 3 |  | 3 |  | 3 |  | 3 |  | 0 |
counter = 6
 | 0 |  | 3 |  | 3 |  | 3 |  | 3 |  | 0 |
counter = 6
 | 0 |  | 3 |  | 3 |  | 3 |  | 3 |  | 3 |  | -1 |  | 0 |
counter = 8
 | 0 |  | 3 |  | 3 |  | 3 |  | 3 |  | 3 |  | -1 |  | 0 |
counter = 8
 | 0 |  | 3 |  | 3 |  | 3 |  | 3 |  | 3 |  | 3 |  | 0 |
counter = 8
 | 0 |  | 3 |  | 3 |  | 3 |  | 3 |  | 3 |  | -1 |  | 0 |
counter = 8
 | 0 |  | 3 |  | 3 |  | 3 |  | 3 |  | 3 |  | -1 |  | 0 |
counter = 8
 ||
counter = 0
```

5 More submission information

After you have submitted your work, check that the process was successful and that you submitted the right file. Before the deadline, you can upload and replace your submission as often as you need through the *Edit submission* button.

Extensions

This assignment is subject to the College policy on extensions. If you believe you need an extension, read the documentation carefully and see the College guidelines here.

Extenuating circumstances

If you submit an assessment and believe that the standard of your work was substantially affected by your current circumstances, you can apply for Extenuating Circumstances. Details on how to apply for this can be found here. Before applying, read carefully the accompanying documentation on the above link. Please note decisions on Extenuating Circumstances are made at the end of the academic year.

Late Submission

In the absence of acceptable extenuating cause, late submissions will be penalised as follows:

- for work submitted up to 24 hours late, the mark will be reduced by ten percentage marks;
- for work submitted more than 24 hours late, the maximum mark will be zero.

Suggestion. If you think you will not be able to meet the deadline, make an on-time submission using 202223 CS2850 CW2: C mini-projectand add the late one to 202223 CS2850 CW2 Late: C mini-project (for late submission only)when your final version is ready. Use the flag `LATE` in the new submission's file name and note that any submission to 202223 CS2850 CW2 Late: C mini-project (for late submission only)will overwrite the original one.

A Implementation of `printList` and `deAllocateBlock`

```
void printList(struct list *pList, int *counter) {
    struct node *cur  = pList->head;
    while (cur) {
        printf(" | %d | ", cur->i);
        cur = cur->next;
    }
    if (!(pList->head) || !(pList->tail) || *counter == 0)
        printf(" ||");
    printf("\ncounter = %d\n", *counter);
}
```

Listing 6: `printList`

```
void deAllocateBlock(struct list *pList, int *counter, int nNodes) {
    int n = (pList->length) - nNodes;
    while ((pList->length) > n) {
        struct node *temp = pList->left->next;
        pList->left->next->next->prev = pList->left;
        pList->left->next = temp->next;
        deAllocator(temp, counter);
        (pList->length)--;
    }
}
```

Listing 7: `deAllocateBlock`

The plain text version of all codes in this document is available here.