Programmering for computerteknologi
Hand-in Assignment Exercises

Week 6: Programming with pointers

Please make sure to submit your solutions by next **Monday** (2023-10-09).

In the beginning of each question, it is described what kind of answer that you are expected to submit. If Text answer AND Code answer is stated, then you need to submit BOTH some argumentation/description and some code; if just Text answer OR Code answer then just some argumentation/description OR code. The final answer to the answers requiring text should be one pdf document with one answer for each text question (or text and code question). Make sure that you have committed your code solutions to your GitHub Repository.

Note: the **Challenge** exercises are *optional*, the others mandatory (i.e. you **have** to hand them in).

Exercises

1)

Text answer Old exam question Consider the following program fragment:

```c
int x;
int y;
int z;
int* w;
int* q;

x = 0;
y = 1;
z = 2;
w = &x;
q = &y;
*w = y;
*q = z;
*w = x + y + z + *q;
*q = x + y + z + *w;

printf("x=%d, y=%d, z=%d\n", x, y, z);
```

? Question

What does the program print when it is executed?

2)

Code answer Write a function `int max(int* numbers, int size)` that, given an array of numbers (and its size), finds the maximum value in the array.

> **Info**
>
> You may assume that the array is not empty (i.e. size > 0).

Include assertions in the implementation of `max` to ensure that the precondition is fulfilled when executing the function.

3)

Text answer Code answer Consider the following program (you can find it in `./list.c`):

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

// NOTE: in the github repository this struct is defined in `./list.h`
typedef struct node {
  int data;
  struct node *next;
} node;

void add(node *head, int x) {
  // pre:  head points to the first, empty element.
  //       The last element's next is NULL
  // post: A new node containing x is added to the end of the list

  assert(head != NULL);
  node *p = head;
  while (p->next != NULL) {
    p = p->next;
  } // p points to the last element

  node *element = malloc(sizeof(node));
  element->next = NULL;
  element->data = x;
  p->next = element;
}

int main() {
  node *list = malloc(sizeof(node));
  list->next = NULL; // create first empty element
  add(list, 1);
  add(list, 3);
  add(list, 2);
  // Show list here
  add(list, 2);
  // Show list here
```

```
37
38    return 0;
39  }
```

(a) Draw **two** diagrams that shows the contents of `list` at the two lines with the comment `// show list here` in `main` (line 34 and 36).

> **Info**
>
> The first element is empty and holds no data. I.e. if I have a list with two elements, it has $2 + 1 = 3$ nodes (the first, empty one and then two nodes holding data)
>
> The same definition is used in all functions.

(b) Implement a function with the following signature: `int size(node *l)`. It has the same precondition as `add` and returns the number of elements in the list. E.g. if `size(list)` was printed out at the first `// show list here`) in main, the result would be 3.

(c) What does the following code do when executed? (i.e. does the code fulfill the post condition? If not, what happens?)

```
1  void printout(node *l) {
2    // pre:  head points to the first, empty element.
3    //       The last element's next is NULL
4    // post: The values of the list are printed out
5    node *p = l->next;
6    while (p != NULL) {
7      printf("%d, ", p->data);
8    }
9    printf("\n");
10 }
```

(d) Correct the function above so the post condition is fulfilled.

(e) Write a function `int largest(node *l)`. The pre- and post conditions are the following:

```
1  // pre:  head poinst to the first, empty element.
2  //       The last element's next is NULL.
3  // post: Returns the largest value of the list
```

4) **Challenge**

(PC-1.6.6) *Interpreter*. A certain computer has **ten** registers and **1000** words of RAM. Each register or RAM location holds a three-digit integer between 0 and 999. Instructions are encoded as three-digit integers and stored in RAM. The encodings are as follows:

Code	Instruction
100	halt
$2dn$	set register d to n (between 0 and 9)
$3dn$	add n to register d
$4dn$	multiply register d by n
$5ds$	set register d to the value of register s
$6ds$	add the value of register s to d
$7ds$	multipy register d by the value of register s
$8da$	set register d to the value in RAM whose address is in register a
$9sa$	set the value in RAM whose address is in register a to that of register s
$0ds$	goto the location in register d unless register s contains 0

All registers initially contain 000. The initial content of the RAM is read from standard input. The first instruction to be executed is at RAM address 0. All results are reduced modulo 1,000.

Input

An input program consists of up to 1,000 three-digit unsigned integers, representing the contents of consecutive RAM locations starting at 0. Unspecified RAM locations are initialised to 000. This is followed by a blank line.

Output

The output is a single integer: the number of instructions executed up to and including the halt instruction. You may assume that the program does halt.

Example

```
299
492
495
399
492
495
399
283
279
689
078
100
000
000
000

16
```

> **Info**
>
> If you quickly want to test your interpreter with different combinations of input, a good idea would be to put your input data into a text file, and then read the contents of the file when your program starts. This way you do not have to recompile your program every time you want to test some other input.
>
> An example is given below of how to read a file of integers into a dynamic array of `int` that you can use for this 😁

```c
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

// A struct that contains a dynamic array of integers
// The struct owns the array and is responsible for freeing it
typedef struct {
  int *buffer;
  size_t size;
  size_t capacity;
} array_of_int;

array_of_int read_int_array_from_file(const char *filename) {
  assert(filename != NULL);
  FILE *file = fopen(filename, "r");
  if (file == NULL) {
    fprintf(stderr, "Could not open file '%s'\n", filename);
    exit(1);
  }

  const size_t initial_capacity = 256;
  size_t capacity = initial_capacity;
  int *array = malloc(capacity * sizeof(int)); // Start with a smaller initial capacity
  size_t size = 0;

  if (array == NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    exit(1);
  }

  while (true) {
    int value = 0;
    int status = fscanf(file, "%d", &value);
    if (status == EOF) {
      break;
    }

    if (size >= capacity) {
      // Double the capacity and reallocate memory
      capacity *= 2;
```

```
      int *new_array = realloc(array, capacity * sizeof(int));
      if (new_array == NULL) {
        fprintf(stderr, "Memory reallocation failed\n");
        free(array);
        exit(1);
      }
      array = new_array;
    }

    array[size] = value; // Insert the value into the array
    size++;
  }

  fclose(file);
  return (array_of_int){.buffer = array, .size = size, .capacity = capacity};
}

int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
    exit(1);
  }
  array_of_int array = read_int_array_from_file(argv[1]);
  for (size_t i = 0; i < array.size; i++) {
    printf("%d\n", array.buffer[i]);
  }
  free(array.buffer);

  return 0;
}
```