1. Pre-Check

This section is designed as a check to allow you to determine whether you understand the concepts covered in class. Answer the following questions and include an explanation:

1.1. True or False: Parameter passing (i.e. when calling functions) is done by value in C.

1.2. What is a pointer in C? What does it have in common with array structures?

1.3. 1.3. If you try to dereference a variable that is not a pointer (i.e. prefix an asterisk to it), what happens? What about when you release it (i.e. `free(…)`)?

2. Data organisation in memory

Consider the data structure type defined below.

```
typedef struct _data {
    char name[13];          // first and last names
    unsigned short age;    // in years, ex: 23
    char gender;            // M: Male, F: Female
    int id[4];              // ex: 1994,408,10,7212
} data;
```

Suppose that an "`employee`" structure of type "`data`" is allocated at memory address "`0x8040`" with the following initializations:

```
data employee = {
    .name = "Tintin Lupin",
    .age = 23,
    .gender = 'M',
    .id = {1994,408,10,7212}
};
```

2.1. If we consider that “`sizeof(char) == 1`, `sizeof(short) == 2`, and `sizeof(int) == 4`”, and if we also consider a memory organization in “little-endian” mode, give the hexadecimal representation of the bytes of the “`employee`” structure in memory.

Address	Data (bytes)							
0x8040								
0x8048								
0x8050								
0x8058								
0x8060								

2.2. The same question as before but using the "big-endian" mode this time.

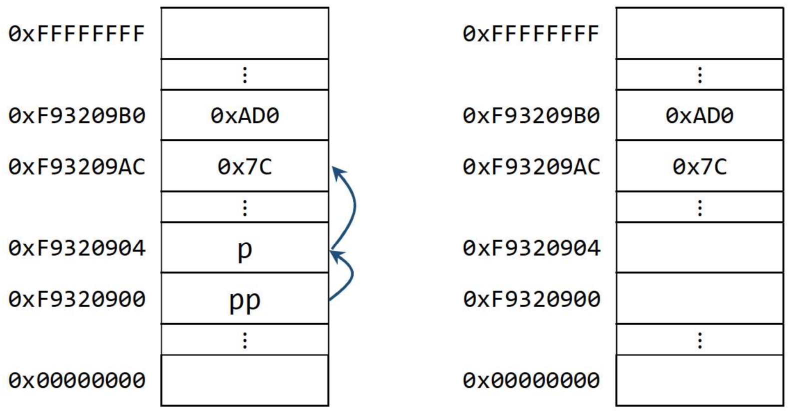
Address	Data (bytes)							
0x8040								
0x8048								
0x8050								
0x8058								
0x8060								

3. Memory in C

The C language is syntactically very similar to Java, but there are some key differences:

- C is "function-oriented" not "object-oriented". So, there are no objects.
- There is no “garbage collector” or automatic memory management in the C language. Dynamic memory allocations and releases are explicitly managed by the programmer (i.e. using `malloc(), …, free()`).
- Pointers are used explicitly in the C language. If `"p"` is a pointer, then `"*p"` indicates (i.e. *points to*) the data to be used and not the value of `"p"` (*i.e.* the memory address). If `"x"` is a variable, then `"&x"` returns the address (*i.e.* a pointer) of `"x"` and not the value of `"x"`.

Below, on the left, a computer memory is represented by a box-and-pointer diagram. The addresses were chosen arbitrarily.



Assume a pointer to an integer (i.e. `int* p`) is allocated at address `0xF9320904`. Let's also assume an integer variable (i.e. `int x`) being allocated at address `0xF93209B0`. From the left diagram above, one can verify:

- `*p` should return the value `0x7C`.
- `p` is assigned the value `0xF93209AC` (i.e. the address where the value `0x7C` is stored).
- `x` contains the value `0xAD0`.
- `&x` will return the value `0xF93209B0` (i.e. the address where "`x`" is stored).

Now assume a pointer to a pointer to an integer (i.e. `int** pp`) is allocated at address `0xF9320900` (see left diagram above).

3.1. What will be the value returned by `pp`? What about `*pp`? and `**pp`?

3.2. Implement the `swap()` function to exchange the values of two integers in memory

```
void swap(int* x, int* y) {



}
```

3.3. Implement `mystrlen()` function that returns the number of bytes in a C string (similar to the standard C library function `strlen()`).

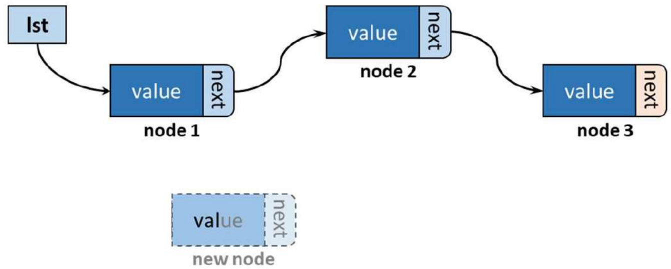
```
int mystrlen(char* str) {




}
```

Let the linked list "`ll_node`" defined as below. Assume as well that argument "`lst`" in exercises 2.3 – 2.4 points to the first element of the linked list (i.e. the head of the list) or contains `NULL` if the list is empty.

```
struct ll_node {
    int value;
    struct ll_node* next;
}
```



Note: The nodes are not necessarily contiguous in memory!

3.4. Write the code for inserting an item at the beginning of the linked list.

```
void insert (struct ll_node** lst, int val ) {

}
```

3.5. Implement the function `release_ll` to empty the entire list

```
void release_ll(struct ll_node * lst) {

}
```

4. Beware of pointers

4.1. Something is wrong with the C code below! Can you spot the problem?

```
int* get_money(int cash) {
  int* money = malloc(2017 * sizeof(int));
  if(!cash)
      money = malloc(1 * sizeof(int));
  return money;
}
```

Review the following functions and fix *any* problems

4.2. Return the total of all elements in the array `summands`

```
int sum(int* summands) {
 int _sum = 0;
 for(int i = 0; i <sizeof(summands); i++)
     _sum += *(summands + i);
 return _sum;
}
```

4.3. Increment the characters of the `string` stored at the beginning of an array of bytes of length `n >= strlen(string)`. MUST NOT modify memory areas outside the character string.

```
void increment(char* string, int n) {
 for(int i = 0; i < n; i++)
     *(string + i)++;

}
```

4.4. Copying the string `src` into `dst`.

```
void copy(char* src, char* dst) {
   while(*dst++ = *src++);

}
```

4.5. Replace, if there is enough space in a character string given as a parameter, with the string `"This course is fantastic!"`. The function should do nothing if the condition is not true.. You may assume that parameter `length` gives the correct length of the `src` string.

```
void ado(char* src, unsigned int length) {
 char *srcptr, replacteptr;
  char remplacement[26] = "This course is fantastic!";
  srcptr = src ;
  replaceptr = replacement;
  if(length >= 26) {
    for(int i=0; i<26; i++)
       *srcptr++ = *replaceptr++;
  }
}
```