Computer Design | Memory and Pointers in C

Computer Science (2^{nd} year B.Sc.) | Seminar #2

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.
TRUE! A copy of the actual arguments is passed to the respective formal arguments. If passing **by reference** is desired, then one must pass the location (i.e. the address) of the actual arguments (i.e. use pointers).

1.2. What is a pointer in C? What does it have in common with array structures?
A pointer is just a sequence of bits interpreted as a memory address. An array in C language acts as a pointer to the first element of that array.

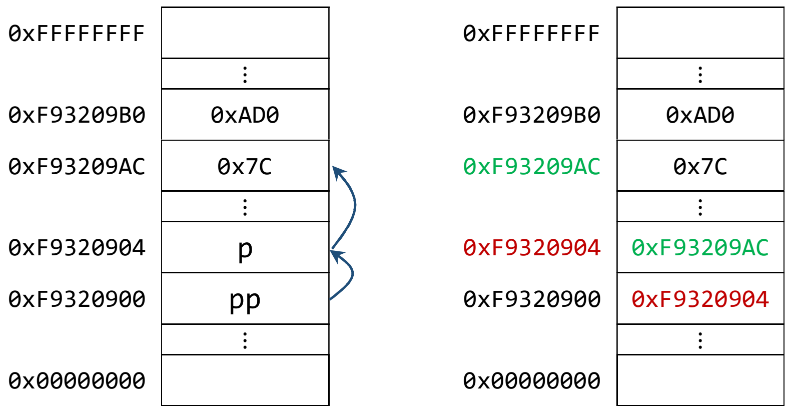
1.3. 1.3. If you try to dereference a variable that is not a pointer (i.e. prefix an asterisk to it), what happens? What about when you release it (i.e. `free(…)`)?
C will treat the bits of this variable as if it were a pointer and attempt to access the "pointed" data. Your program will most likely exit with the "`memory segfault`" error. If you free a variable that has been previously freed or not allocated, your program's behaviour will be undefined and exit with an "`invalid free`" error.

2. 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"`.

In the following example, on the left, a computer memory is represented by a box-and-pointer diagram. On the right, we see how this memory is actually organized in the computer (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`. As can be seen 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).

2.1. What will be the value returned by `pp`? What about `*pp`? and `**pp`?
pp has value `0xF9320904`. *pp will return `0xF93209AC`, and **pp will give `0x7C`.

2.2. 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)); // "memory leak" if !cash
  return money;
}
```

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

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

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

```
void insert (struct ll_node** lst, int val ) {
   struct ll_node* elem = (struct ll_node*) malloc(sizeof(struct ll_node));
   elem->value = val;
   elem->next = *lst;
   *lst = elem;
}
```

2.4. Implement the function `release_ll` to release/empty the entire list

```
void release_ll(struct ll_node * lst) {
// recursive solution
   if(lst != NULL) {
      release_ll(lst->next);
      free(lst);
   }

// or iterative solution
   while (lst) {
      struct ll_node *temp = lst->next;
      free(lst);
      lst = temp;
   }
}
```

3. Programming with pointers

Implement the following functions so that they work as described.

3.1. A function that allows you to swap the values of two integers given as parameters.

```
void swap(int* x, int* y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}
```

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

```
int mastrlen(char* str) {
  int compte = 0;
  while(*str++) {
    compte++;
  }
  return compte;
}
```

Review the following functions and fix *any* problems

3.3. Return the total of all elements in the array summands

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

3.4. 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) {  // parameter n is not needed
 for(int i = 0; i < n; i++)  // for(int i = 0; string[i] != 0; i++)
    *(string + i)++;   // string[i]++;
                       // ou (*(string + i))++;
}
```

3.5. Copying the string src into dst.

```
void copy(char* src, char* dst) {
   while(*dst++ = *src++);
   // there are no errors in this code
}
```

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

4. How Data is stored 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 sexe;             // 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,
    .sexe = 'M',
    .id = {1994,408,10,7212}
};
```

4.1. 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	54	69	6E	74	69	6E	20	4C
0x8048	75	70	69	6E	00	-	17	00
0x8050	4D	-	-	-	CA	07	00	00
0x8058	98	01	00	00	0A	00	00	00
0x8060	2C	1C	00	00				

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

Address	Data (bytes)							
0x8040	54	69	6E	74	69	6E	20	4C
0x8048	75	70	69	6E	00	-	00	17
0x8050	4D	-	-	-	00	00	07	CA
0x8058	00	00	01	98	00	00	00	0A
0x8060	00	00	1C	2C				

Remember that, for performance reasons (we will see later in the course) data is (always) stored at memory addresses that are multiple of their sizes. On the other hand, a C string is always terminated with an implicit zero – don't forget to count it when you allocate memory space for the string!