Master IOT & AI4CI – Bash & C

Lab Handout: Dynamic Memory Management and Pointers in C

Sami Taktak
sami.taktak@cnam.fr

2025

During the laboratory sessions we will use a GNU/Linux operating system.

Learning Objectives

By the end of this lab, you will be able to:

1. Understand and use pointers in C
2. Differentiate between stack, static, and heap memory allocation
3. Use `malloc`, `realloc`, and `free` correctly
4. Detect and fix memory-related bugs (leaks, dangling pointers, buffer overflows)
5. Apply dynamic memory management to implement real-world data structures

Background

In C, memory management is manual: you, the programmer, must explicitly allocate and free memory.

- **Stack**: Automatically managed, limited lifetime (local variables)
- **Static**: Fixed size, exists throughout program execution
- **Heap**: Dynamically managed, controlled using `malloc`, `realloc`, and `free`

Common mistakes include:

- **Memory leaks**: Forgetting to free allocated memory
- **Dangling pointers**: Using memory after it is freed
- **Buffer overflows**: Writing outside allocated bounds

Part 1: Pointer Basics

a) Swapping integers using pointers

```c
#include <stdio.h>

void swap(int *a, int *b) {
    // TODO: implement swap using a temporary variable
}

int main() {
    int x = 5, y = 10;
    printf("Before swap: x=%d, y=%d\n", x, y);
    swap(&x, &y);
    printf("After swap: x=%d, y=%d\n", x, y);
    return 0;
}
```

Task: Implement `swap` and test.

b) Reversing an array using pointers

```c
#include <stdio.h>

void reverse(int *arr, int n) {
    // TODO: reverse array in place using two pointers
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = 5;
    reverse(arr, n);

    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Task: Implement `reverse` using pointer arithmetic.

Part 2: Dynamic Memory Allocation

a) Allocating an integer array with `malloc`

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

```

```c
#define SIZE 256

int main() {
    char str[SIZE];
    int n;

    printf("Enter number of elements: ");
    fgets(str, SIZE, stdin);
    n = atoi(str);

    // TODO: allocate array dynamically using malloc
    // TODO: initialize with squares of indices
    // TODO: print the array

    // TODO: free memory
    return 0;
}
```

Task: Allocate, initialize, print, and free.

b) Resizing with `realloc`

Modify exercise Part 2:-a):

- Ask user for a new size
- Use `realloc` to resize the array
- Fill new slots with cubes of indices
- Print final array

c) Dynamic String Allocation

Task: Modify the following program so that it reads as many strings as the user provides (ends on empty line).

The length of strings are unknown. Memory space for strings should grows as the string is read.

Read strings are stored in a dynamic array of strings.

The program should:

- Start with a string array of 2 strings
- Start with a string buffer of 5 chars
- Reallocate the string buffer when it is full
- Read characters until newline
- Read the next string until a empty string is read
- Reallocate the string array when it is full
- Print the final string array

```c
#include <stdio.h>

#define SIZE 5

int main() {
  char tab[SIZE+1];

  int i = 0;

  printf("Give %d characters followed by Enter: ", SIZE);
  while( (i < SIZE) ) {
    char c = getchar();
    if (c == '\n' ) break;
    tab[i] = c;
    i++;
  }
  tab[i]= '\0';

  if (i == SIZE ) while ( getchar() != '\n');

  printf("%s\n", tab);

  return 0;
}
```