Object Oriented programming and software engineering – Lab 3

Adam Korytowski - 2025

1. Pointers

Pointers in C++ store the addresses of variables and allow operations on them. With pointers, we can dynamically allocate memory, pass variables to functions by reference, and create data structures such as lists or trees.

- Analyze the following code and expect what happens in it. Then modify it so that the pointer *ptr* changes the value of the variable *a* (1 point)

```cpp
#include <iostream>
using namespace std;

int main()
{
    int a = 10;
    int* ptr = &a; // Pointer stores the address of variable a

    cout << "Value of a: " << a << endl;
    cout << "Address of a: " << &a << endl;
    cout << "Value of pointer ptr: " << ptr << endl;
    cout << "Value pointed to by ptr: " << *ptr << endl;

    return 0;
}
```

2. Pointers and arrays

In the following situation:

```cpp
int* ptr;
int tab[5] = { 1,2,3,4,5 };
ptr = &tab[3];
```

if during program we want the pointer to point to another element of this array, use the following statement

ptr += n;

- Modify the code so that the pointer points to value of the array in index [1] (1 point)

- Create another pointer that points to the beginning of the array, without using indexing operator (1 point)

3. Dynamic memory allocation

In C++, memory allocation allows you to dynamically manage memory at runtime instead of relying on fixed-size variables or arrays. This provides flexibility, efficient memory usage, and enables advanced data structures.

Why to use memory allocation?

Flexibility in Managing Memory

- ✓ Static arrays (e.g., int arr[100];) have a fixed size at compile time, which can lead to wasted memory or limitations.
- ✓ Dynamic memory allows you to allocate only as much memory as needed during execution.

Efficient Memory Usage

- ✓ If you allocate too much memory statically, you waste unused space.
- ✓ If you allocate too little, the program runs out of memory.
- ✓ Dynamic allocation uses only what's needed and releases it when no longer necessary.

Passing Large Data Between Functions Efficiently

- ✓ When passing large objects to functions, passing by pointer is more efficient than copying them.
- ✓ This avoids stack overflow and improves performance.

To reserve memory areas we will use the *new* operator (in c, the equivalent of malloc). To clear the area delete is used (equivalent to c- free).

Example:

```cpp
// Dynamically allocate an integer
int* ptr = new int;

// Assign a value
*ptr = 42;

// Print the value and address
cout << "Value: " << *ptr << endl;
cout << "Memory address: " << ptr << endl;

// Deallocate memory
delete ptr;

// Avoid dangling pointer
ptr = nullptr;
```

4. Additional tasks

- Learn how to allocate, modify, and free a dynamically allocated array in C++ (2 pts)
    - Dynamically allocate an array of n integers using *new*.
    - Let the user input values into the array.
    - Print the array using pointer arithmetic.
    - Free the allocated memory using *delete*[].

- Understand how to use pointers to manipulate variables directly (2 pts)
    - Write a function swapNumbers(int* a, int* b) that swaps two integers using pointers.
    - In main(), declare two integers, input their values, and call the function to swap them.
    - Print the values before and after the swap.