МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Кафедра комп'ютерної інженерії

Дисципліна: Програмування
Спеціальність: 123 «Комп’ютерна інженерія»
Освітній рівень: Перший (бакалаврський)
Семестр: I

ЛАБОРАТОРНА РОБОТА № 6

Тема: Динамічне виділення пам'яті в купі (Heap), масштабування буферів через `realloc()`, діагностика витоків пам'яті за допомогою AddressSanitizer та керування історією змін у Git

---

1. МЕТА РОБОТИ

- Набути практичних навичок динамічного виділення пам'яті в купі (Heap) під час виконання програми за допомогою підпрограм `malloc()` та `calloc()`.
- Опанувати безпечний інженерний патерн масштабування пам'яті через підпрограму `realloc()` із використанням тимчасового покажчика для захисту від безповоротних витоків пам'яті.
- Засвоїти культуру повного життєвого циклу ресурсів: своєчасне звільнення виділених блоків через `free()` та обов'язкове примусове занулення покажчиків (`ptr = NULL`) для усунення загрози висячих адрес (Dangling Pointers).
- Опанувати інструменти автоматичного виявлення апаратних дефектів пам'яті за допомогою компіляторного аналізатора **AddressSanitizer (ASan)**: виявлення витоків пам'яті (Memory Leaks), подвійного звільнення (Double Free) та звернення до пам'яті після звільнення (Use-After-Free).
- Опанувати навігацію та керування історією змін у системі Git: дослідження різниці у файлах через `git diff`, скасування незафіксованих змін через `git restore` та безпечне скасування комітів за допомогою інверсного коміту `git revert`.
- Підготувати підсумкову інженерну документацію та долучити **обов'язкову пояснювальну записку у форматі PDF** до звітності.

2. НЕОБХІДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

1. Текстовий редактор / середовище розробки: **Visual Studio Code** (із розширенням *C/C++* від Microsoft) або **CLion**.
2. Компілятор: **GCC** (GNU Compiler Collection) з підтримкою прапорців санітайзера `-fsanitize=address`.
3. Система контролю версій: **Git**.
4. Програма для підготовки технічної документації та експорту в PDF (**LaTeX**, **MS Word** або **LibreOffice Writer**).

---

3. ПОСЛІДОВНІСТЬ ВИКОНАННЯ РОБОТИ

Етап 0. Керування історією та скасування змін у Git

1. Відкрийте термінал, перейдіть до кореневого каталогу вашого репозиторію та створіть нову гілку `lab06`:

```
cd ~/projects/programming-sem1
git checkout -b lab06
# або: git switch -c lab06
mkdir lab06
cd lab06
```

2. Створіть тренувальний файл `history_test.txt` із текстом: `Стабільна конфігурація периферії v1.0`. Зафіксуйте його:

```
git add history_test.txt
git commit -m "feat(lab6): add initial configuration for history test"
```

3. **Практика роботи з `git diff` та `git restore`:**
   - Відкрийте `history_test.txt`, додайте помилковий рядок: `Некоректний експериментальний параметр` і збережіть файл (НЕ робіть коміт!).
   - Перегляньте незафіксовану різницю в терміналі:

```
git diff history_test.txt
```

   - Скасуйте незафіксовані зміни, повернувши робочий файл до чистого стану останнього коміту:

```
git restore history_test.txt
```

   - Переконайтеся, що файл знову містить лише початковий стабільний рядок.

4. **Практика безпечного скасування коміту через `git revert`:**
   - Додайте у файл рядок: `Помилкове оновлення драйвера шини`, збережіть і закомітьте:
     ```
     git add history_test.txt
     git commit -m "feat(lab6): add erroneous driver update"
     ```
   - Перегляньте хеш останнього коміту: `git log --oneline -n 3`.
   - Скасуйте дію цього коміту інженерним шляхом — створенням компенсуючого зворотного коміту:
     ```
     git revert HEAD --no-edit
     ```
   - Перевірте історію через `git log --oneline`: переконайтеся, що з'явився новий коміт скасування, а помилковий рядок автоматично зник із файлу.

Етап 1. Виділення динамічного масиву з контролем на NULL (`task1.c`)

1. У каталозі `lab06` створіть файл `task1.c`.
2. Відповідно до свого номера варіанта з **Таблиці 1** реалізуйте програму:
   - Програма запитує у користувача необхідний розмір масиву N під час виконання (Runtime).
   - Виділяє динамічний блок пам'яті під N елементів за допомогою підпрограми `malloc()` або `calloc()` (розмір блоку обов'язково розраховується за допомогою оператора `sizeof`).
   - **Обов'язковий захист (Null-Check):** Якщо повернений покажчик дорівнює `NULL` — програма виводить аварійне повідомлення про дефіцит RAM і коректно завершує роботу з кодом помилки.
   - Заповнює масив значеннями та виконує індивідуальне обчислення згідно з варіантом.
   - **Очищення ресурсів:** Звільняє виділену пам'ять підпрограмою `free()`, після чого **негайно записує в покажчик значення** `NULL`!
3. Скомпілюйте та зафіксуйте код у Git:
   ```
   gcc -Wall -Wextra task1.c -o task1
   git add task1.c
   git commit -m "feat(lab6): implement dynamic 1D array with null-checks"
   ```

Етап 2. Потокове масштабування буфера через `realloc()` (`task2.c`)

1. У каталозі `lab06` створіть файл `task2.c`.
2. Відповідно до свого номера варіанта з **Таблиці 2** реалізуйте алгоритм динамічного розширення буфера пам'яті:
   - Програма стартує з виділення малого базового буфера (наприклад, на 2 чи 4 елементи).
   - Дані надходять від користувача або генератора по одному елементу за раз у циклі до введення зазначеного у варіанті термінального символу/числа.
   - Коли зарезервований буфер заповнюється, програма динамічно збільшує його місткість за допомогою `realloc()`.
   - **Захисний патерн тимчасової адреси:**
     - Результат роботи `realloc()` обов'язково приймається в **окремий тимчасовий покажчик**!
     - Перевіряється умова: якщо тимчасовий покажчик дорівнює `NULL` — програма повідомляє про помилку дефіциту ресурсів, звільняє старий збережений буфер і безпечно виходить, запобігаючи витоку пам'яті.
     - Лише після підтвердження успішного виділення основний робочий покажчик оновлюється новою адресою.
   - Після зупинки введення вивести всі накопичені дані, їхню фінальну кількість та коректно звільнити пам'ять за допомогою `free()`.
3. Зафіксуйте код у Git:

```
git add task2.c
git commit -m "feat(lab6): implement safe dynamic array reallocation
pattern"
```

Етап 3. Аудит витоків пам'яті через AddressSanitizer (`task3.c`)

1. У каталозі `lab06` створіть файл `task3.c`.
2. Відповідно до свого номера варіанта з **Таблиці 3** напишіть програму створення та обробки складної динамічної структури в купі (матриці, буфера пакетів чи масиву рядків).
3. Скомпілюйте код із прапорцями апаратного санітайзера пам'яті:

```
gcc -Wall -Wextra -fsanitize=address -g task3.c -o task3_asan
```

4. **Проведення експерименту з фіксацією логів:**

- Навмисно закоментуйте рядок виклику підпрограми `free()` для одного з виділених блоків і запустіть програму `./task3_asan`.
- Зафіксуйте червоний звіт AddressSanitizer: `ERROR: LeakSanitizer: detected memory leaks` із зазначенням точного розміру втраченої пам'яті.
- Відновіть правильне звільнення всіх блоків пам'яті (включно із зануленням покажчиків), перекомпільуйте програму та переконайтеся у повній відсутності помилок витоку при повторному запуску.

5. Зафіксуйте код у Git:

```
git add task3.c
git commit -m "feat(lab6): verify zero memory leaks using AddressSanitizer"
```

Етап 4. Долучення звіту та фінальний Push

1. Підготуйте інженерну пояснювальну записку та збережіть її **виключно у форматі PDF** із точною назвою `Report_Lab06.pdf` у каталозі `lab06`.
2. Додайте файл звіту до Git, закомітьте, перемкніться на гілку `main` та злийте напрацювання:

```
git add Report_Lab06.pdf
git commit -m "docs(lab6): add mandatory engineering PDF report"
git checkout main
git merge lab06
git push origin main
```

---

4. ІНДИВІДУАЛЬНІ ВАРІАНТИ ЗАВДАНЬ

Номер варіанта визначається номером у списку академічного журналу групи:

$$\text{Варіант} = (\text{Номер у списку} - 1) \pmod{15} + 1$$

Таблиця 1. Варіанти до Завдання 1 (Динамічний одновимірний масив)

Вар.	Тип даних та індивідуальне обчислення над динамічним масивом розміром N
1	Тип `double`. Виділити через `malloc`. Знайти середнє геометричне додатних елементів масиву.

Вар.	Тип даних та індивідуальне обчислення над динамічним масивом розміром N
2	Тип `int`. Виділити через `calloc`. Знайти кількість елементів, які діляться на перший елемент без залишку.
3	Тип `float`. Виділити через `malloc`. Обчислити різницю між максимальним та мінімальним значеннями.
4	Тип `long`. Виділити через `calloc`. Знайти суму елементів із непарними порядковими номерами.
5	Тип `double`. Виділити через `malloc`. Знайти індекс першого елемента, що перевищує задане користувачем число.
6	Тип `short`. Виділити через `calloc`. Порахувати кількість локальних мінімумів (елементів, менших за обох сусідів).
7	Тип `int`. Виділити через `malloc`. Виконати циклічний зсув динамічного масиву праворуч на K позицій.
8	Тип `float`. Виділити через `calloc`. Знайти добуток елементів, розташованих після найменшого за модулем числа.
9	Тип `double`. Виділити через `malloc`. Замінити всі від'ємні елементи середнім арифметичним значенням усього масиву.
10	Тип `int`. Виділити через `calloc`. Знайти другий за величиною додатний елемент у вибірці.
11	Тип `float`. Виділити через `malloc`. Порахувати кількість елементів, які потрапляють у діапазон $[A; B]$.
12	Тип `double`. Виділити через `calloc`. Знайти номер найдовшої послідовності додатних чисел, що йдуть підряд.
13	Тип `int`. Виділити через `malloc`. Переставити елементи масиву так, щоб спочатку йшли всі парні, а потім непарні.
14	Тип `short`. Виділити через `calloc`. Обчислити суму абсолютних різниць між сусідніми елементами масиву.
15	Тип `double`. Виділити через `malloc`. Визначити, чи впорядкований масив за суворим зростанням значень.

Таблиця 2. Варіанти до Завдання 2 (Масштабування через `realloc()`)

Вар.	Стратегія та логіка динамічного розширення буфера потокових даних
1	Початковий розмір: 2 цілих числа. При переповненні розмір подвоюється. Ознака зупинки: число -999.
2	Початковий розмір: 5 дійсних чисел (`double`). При переповненні додається $+5$ елементів. Ознака зупинки: від'ємне число.
3	Початковий розмір: 3 цілих числа. При переповненні розмір подвоюється. Ознака зупинки: парне число, що ділиться на 10.
4	Початковий розмір: 4 числа (`float`). При переповненні додається $+4$ елементи. Ознака зупинки: введення числа 0.
5	Початковий розмір: 1 ціле число. При надходженні кожного нового елемента буфер зростає рівно на $+1$. Зупинка: число 100.
6	Початковий розмір: 8 цілих чисел. При переповненні розмір збільшується в 1.5 рази. Ознака зупинки: число -1.
7	Початковий розмір: 2 числа (`double`). При переповненні розмір збільшується втричі. Ознака зупинки: число 0.0.
8	Початковий розмір: 6 цілих чисел. При переповненні додається $+3$ елементи. Ознака зупинки: число більше 1000.
9	Початковий розмір: 4 цілих числа. При переповненні подвоюється. Ознака зупинки: послідовне введення двох однакових чисел.
10	Початковий розмір: 5 чисел (`float`). При переповненні додається $+5$ елементів. Ознака зупинки: від'ємне число.
11	Початковий розмір: 3 числа (`double`). При переповненні розмір подвоюється. Ознака зупинки: число, менше за 0.001.
12	Початковий розмір: 2 цілих числа. При переповненні розмір збільшується на $+4$. Ознака зупинки: число -777.
13	Початковий розмір: 4 цілих числа. При переповненні розмір подвоюється. Ознака зупинки: число 0.
14	Початковий розмір: 5 чисел (`short`). При переповненні додається $+10$ елементів. Ознака зупинки: число, кратне 5.
15	Початковий розмір: 2 числа (`double`). При переповненні розмір подвоюється. Ознака зупинки: введення трьох від'ємних чисел підряд.

Таблиця 3. Варіанти до Завдання 3 (Аудит динамічних 2D-структур у купі)

Вар.	Складна динамічна структура для виділення, тестування з ASan та повного очищення
1	Динамічна прямокутна матриця $R \times C$ типу `int`: виділяється як масив покажчиків на окремі динамічні рядки.
2	Ступінчастий (зубчастий) масив трикутника Паскаля: кожен наступний рядок у купі має довжину на 1 більшу за попередній.
3	Динамічний масив текстових рядків: виділити масив покажчиків на N слів, виділити пам'ять під кожне слово, звільнити по черзі.
4	Динамічна матриця $N \times N$ типу `double`, виділена як суцільний плоский монолітний блок у купі з 2D-адресацією через формулу.
5	Динамічний масив буферів сенсорів: K датчиків, кожен із яких має власний динамічний масив вимірювань довільної довжини.
6	Динамічна симетрична матриця $N \times N$: пам'ять у купі виділяється виключно під елементи на і нижче головної діагоналі.
7	Динамічний масив цілих чисел, який у середині роботи дублюється у новий динамічний масив-копію, після чого обидва коректно очищуються.
8	Динамічна таблиця зв'язків: масив із N покажчиків, кожен із яких вказує на динамічний список сусідніх вузлів графа.
9	Динамічний буфер пакетів: матриця байтів $M \times 64$, де кожен рядок імітує мережевий Ethernet-пакет у купі.
10	Динамічна розріджена матриця: вектор покажчиків лише на ті рядки, які містять хоча б одне ненульове значення.
11	Динамічний буфер аудіосемплів: два паралельні динамічні масиви типу `float` для лівого та правого каналів звуку.
12	Масив покажчиків на динамічні рядки тексту з клавіатури, де кількість слів і довжина кожного слова визначаються в Runtime.
13	Динамічна тривимірна сітка простору $X \times Y \times Z$ типу `int`, реалізована як масив покажчиків на матриці.
14	Динамічний стек цілих чисел на базі динамічного масиву: функції додавання `push` та видалення `pop` із контролем пам'яті.
15	Динамічний циклічний кільцевий буфер подій у купі з автоматичним розширенням при переповненні.

---

5. ВИМОГИ ДО ЗВІТНОСТІ ТА ЗДАЧІ РОБОТИ

Робота захищається очно в комп'ютерному класі.
Обов'язковою умовою допуску до очного захисту є наявність у репозиторії на GitHub **пояснювальної записки у форматі PDF** (`Report_Lab06.pdf`), яка містить титульний аркуш, індивідуальні формулювання завдань, опис логіки безпечного `realloc` , повні лістинги програм `task1.c` , `task2.c` , `task3.c` , скріншоти тестування (включно зі звітом AddressSanitizer про витік пам'яті та його успішне виправлення) і висновки.

Структура репозиторію на GitHub перед захистом повинна мати вигляд:

```
programming-sem1/
├── .gitignore
├── README.md
├── history_test.txt
├── lab01/
├── lab02/
├── lab03/
├── lab04/
├── lab05/
└── lab06/
    ├── task1.c
    ├── task2.c
    ├── task3.c
    └── Report_Lab06.pdf
```

Примітка щодо Git: Викладач перевіряє історію комітів за допомогою команди `git log --graph --oneline` на предмет наявності чітко зафіксованого коміту скасування змін через команду `git revert` .

---

6. КОНТРОЛЬНІ ЗАПИТАННЯ ДЛЯ ОЧНОГО ЗАХИСТУ

1. У чому полягає фундаментальна різниця між виділенням локального масиву на стеку та виділенням блоку в купі за допомогою `malloc()` ? Яка різниця у швидкодії на рівні процесора?
2. Чим стан пам'яті після виклику `malloc()` відрізняється від стану пам'яті після виклику `calloc()` ?
3. Чому спроба записати результат роботи `realloc()` прямо в той самий покажчик (`ptr = realloc(ptr, new_size)`) вважається грубою інженерною помилкою?
4. Звідки підпрограма `free(ptr)` дізнається, скільки саме байтів пам'яті потрібно повернути системі, якщо ми передаємо їй виключно початкову адресу?

5. Що таке висячий покажчик (Dangling Pointer) і чому запис значення `NULL` у покажчик одразу після виклику `free()` є обов'язковим стандартом безпеки?
6. Що таке витік пам'яті (Memory Leak)? До яких апаратних наслідків призводить регулярний витік пам'яті в довготривалих процесах (серверах, суперциклах мікроконтролерів)?
7. Як працює компіляторний інструмент AddressSanitizer (`-fsanitize=address`) і які типи помилок роботи з пам'яттю він здатен виявити?
8. Чим команда безпечного скасування змін `git revert` відрізняється від команди `git reset --hard` і чому в спільних командних репозиторіях слід обирати саме `revert` ?