Міністерство освіти і науки України
Чернівецький національний університет
імені Юрія Федьковича

Г.П. Івасюк, Т.М. Фратавчан, Р.В. Івасюк

Програмування

Частина 2

Навчальний посібник

Чернівці
Чернівецький національний університет
2025

УДК 044.432.021(075.8)
П 784

Друкується за ухвалою Вченої Ради факультету математики та інформатики Чернівецького національного університету імені Юрія Федьковича, протокол №10 від 23.04.2025 р.

Рецензенти: *В.В.Лазорик*, кандидат фіз.-мат. наук, доцент кафедри математичних проблем управління і кібернетики Навчально-наукового інституту фізико-технічних та комп'ютерних наук Чернівецького національного університету імені Юрія Федьковича

І.П.Мединський, доктор фіз.-мат. наук, професор кафедри прикладної математики Інституту прикладної математики та фундаментальних наук Національного університету «Львівська політехніка»

П 784 **Програмування.** Частина 2. Навчальний посібник / Укл.: Івасюк Г.П., Фратавчан Т.М., Івасюк Р.В. Чернівці: Чернівецький нац. ун-т, 2025. 164 с.

Посібник містить теоретичний матеріал, приклади розв'язування типових завдань, підбір завдань для практичних занять з програмування. Для здобувачів вищої освіти першого (бакалаврського) рівня спеціальностей 122 «Комп'ютені науки» та 124 «Системний аналіз»

УДК 044.432.021(075.8)
© Івасюк Г.П., Фратавчан Т.М., 2025
© Чернівецький національний університет, 2025

Передмова

Ознайомитися з сучасним програмуванням неможливо без розуміння ефективного використання пам’яті, структур даних і алгоритмів. Основу цього становлять вказівники, функції, структури, а також алгоритми сортування, пошуку й робота з файлами. Ці елементи є базовими для вивчення мови програмування С і мов, що базуються на ній.

Цей посібник призначений для студентів, викладачів, а також усіх, хто прагне поглибити свої знання у сфері програмування. Матеріал викладено у логічній послідовності: від основ роботи з пам’яттю та вказівниками до складніших тем. Кожен розділ містить теоретичний виклад матеріалу, питання для самоконтролю та практичні завдання, що дозволяють закріпити отримані знання на практиці.

Окрему увагу приділено рекурсії, динамічному виділенню пам’яті, а також передачі параметрів у функції, оскільки ці теми часто викликають труднощі у початківців. Завдяки прикладам і завданням читачі зможуть не лише зрозуміти базові концепції, а й навчитися застосовувати їх у реальних проєктах.

Автори щиро вдячні своїм вчителям та колегам, які формували їх знання та навички в області викладання інформатики та програмування, формування загальної педагогічної культури: Карабановій Тетяні Петрівні, Фратавчану Валерію Григоровичу, а також своїм колегам – співробітникам кафедри математичного моделювання Чернівецького національного університету імені Юрія Федьковича за щирі поради при підготовці посібника.

Сподіваємося, що цей навчальний посібник буде корисним помічником в освоєнні мови програмування С. Бажаємо успіхів у вивченні матеріалу та вдосконаленні навичок програмування!

Зміст

Передмова 3

Вступ 8

1. Вказівники 10

1.1. Поняття вказівника 10

1.2. Операції з вказівниками 11

1.3. Динамічне виділення пам’яті 15

1.4. Одновимірні динамічні масиви 18

1.5. Двовимірні динамічні масиви 22

Питання для самоконтролю 29

Практичні завдання 31

2. Функції користувача 36

2.1. Оголошення та виклик функцій 36

2.2. Передача параметрів функції за значенням 37

2.3. Передача параметрів функції за адресою 37

2.4. Передача масивів у функції 38

2.5. Рекурсивні функції та їх застосування 38

Питання для самоконтролю 43

Практичні завдання 45

3. Структуровані типи даних 56

3.1. Структури: оголошення та використання 56

3.2. Масиви структур 57

3.3. Передача структур у функції 60
3.4. Вкладені структури 62
Питання для самоконтролю 66
Практичні завдання 67
4. Користувацькі типи даних 77
4.1. Поняття перелічуваних типів (enum) 77
4.2. Об'єднання (union) як структура даних в мові С 81
4.3. Бітові поля у структурах: опис та застосування 84
Питання для самоконтролю 87
Практичні завдання 88
5. Робота з файлами 99
5.1. Основи роботи з файлами 99
5.2. Операції читання та запису 100
5.3. Робота з вказівником на файл 102
5.4. Бінарні та текстові файли 108
Питання для самоконтролю 112
Практичні завдання 113
6. Алгоритми сортування 116
6.1. Бульбашкове сортування 116
6.2. Сортування вибором 118
6.3. Сортування вставкою 120
6.4. Швидке сортування 122

6.5. Сортування злиттям 124
6.6. Сортування підрахунком 127
6.7. Цифрове сортування..... 133
Питання для самоконтролю..... 137
Практичні завдання 138
7. Алгоритми пошуку 143
7.1. Лінійний пошук 143
7.2. Бінарний пошук 144
7.3. Рекурсивний бінарний пошук 146
Питання для самоконтролю..... 149
Практичні завдання 151
Додаток А. Бітові операції..... 157
А.1. Побітові оператори..... 157
А.2. Маскування бітів..... 158
А.3. Перемикання, встановлення та очищення бітів 159
Додаток Б. Таблиця ASCII кодів символів 161
Література..... 162

Вступ

У сучасному світі цифрових технологій програмування стало не лише інструментом створення програмного забезпечення, а й невід'ємною частиною багатьох галузей — від техніки й медицини до економіки та освіти. Уміння мислити як програміст, тобто аналізувати проблему, знаходити для неї оптимальне рішення й реалізовувати його у вигляді чітких інструкцій для комп'ютера, стало ключовою навичкою XXI століття.

Посібник створено з метою надати читачеві систематизовані знання з основ програмування, розвинути логічне мислення, вміння працювати з алгоритмами та структурами даних, а також сформувати практичні навички написання ефективного й читабельного коду. Матеріал орієнтований насамперед на студентів перших курсів технічних спеціальностей, але може бути корисним також для старшокласників, які цікавляться ІТ, або для всіх охочих оволодіти базовими поняттями програмування.

Підручник побудований на принципі поступового ускладнення: кожен розділ логічно продовжує попередній, вводячи нові поняття та прийоми, що ґрунтуються на вже засвоєному матеріалі. Починаючи з теми «Вказівники» ми розглянемо, як розміщуються дані в оперативній пам'яті комп'ютера, як можна ними ефективно керувати, як створювати динамічні структури даних. Далі перейдемо до вивчення теми «Функції користувача» — одного з основних механізмів організації коду. Особливу увагу приділено рекурсії, що є важливим інструментом для розв'язання задач, де рішення зводиться до аналогічних підзадач.

У розділі, присвяченому структурованим типам даних, ви дізнаєтеся, як об’єднувати дані в логічно цілісні одиниці, передавати їх у функції, створювати масиви структур та вкладені структури. Далі ми познайомимося з перелічуваними типами (enum), які дають змогу наочно й зручно описувати множини сталих значень. Не залишено поза увагою й роботу з файлами — важливий етап при зберіганні та обробці великих обсягів інформації. Завершується підручник оглядом основних алгоритмів сортування та пошуку — фундаментальних інструментів для обробки даних.

Кожна тема супроводжується питаннями для самоконтролю, які допоможуть перевірити розуміння ключових понять, а також практичними завданнями, що спрямовані на формування стійких навичок програмування. Значна частина вправ має прикладний характер і може бути легко адаптована до реальних задач.

Навчальний матеріал орієнтований на мову програмування С, яка традиційно вважається однією з найкращих для вивчення базових принципів — зокрема через її низькорівневу природу, прозоре управління пам’яттю та близькість до архітектури комп’ютера. Розуміння програмування на С формує міцну базу для подальшого вивчення сучасних мов, таких як C++, C#, Java, Python та інших.

1. Вказівники

1.1. Поняття вказівника

Вказівник — це спеціальна змінна, яка зберігає адресу іншої змінної в пам’яті. Типи вказівників є важливими, оскільки вони визначають, до якого типу даних належить змінна, на яку вказує вказівник.

Вказівники оголошуються за допомогою символу ***** . Наприклад,

int * — тип вказівника на ціле число (int);

float * — тип вказівника на число з плаваючою комою (float);

double* — тип вказівника на число з подвійною точністю (double);

char * — тип вказівника на символ (char).

Кожен із цих типів вказівників дозволяє працювати з відповідним типом даних, на які вони вказують. Наприклад,

```
int * ptr;
int a = 5;
ptr = &a;
```

змінна *ptr* — вказівник на змінну типу int, вона містить адресу області пам’яті, за якою зберігається ціле число *a* (див. Рис. 1.1).

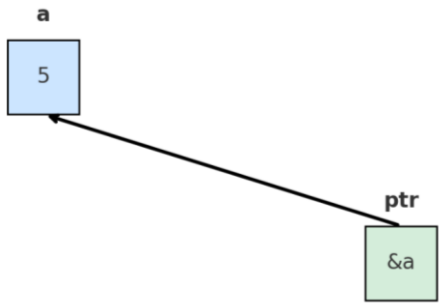


Рис.1.1. Вказівник на змінну

У мові С важливо розуміти, що вказівник сам по собі не створює змінну або пам’ять для зберігання даних — він лише зберігає адресу

іншої змінної. Для отримання адреси змінної використовують **оператор** &. Вираз *ptr = &a* означає, що вказівник *ptr* зберігає адресу змінної *a*.

Розіменування — це процес звернення до значення, яке зберігається за адресою, на яку вказує вказівник. Для цього використовується **оператор розіменування** *, який ставиться перед іменем вказівника.

Наприклад:

```
int value = *ptr;
```

У цьому випадку **ptr* означає "отримати значення, яке зберігається за адресою, що міститься в *ptr*". Якщо *ptr* вказує на змінну *a*, тоді **ptr* еквівалентне *a*. Вираз *value = *ptr* скопіює значення змінної *a* в нову змінну *value*. Також можна використовувати розіменування для зміни значення, наприклад, **ptr = 25*. У цьому випадку значення змінної *a* зміниться на 25, оскільки *ptr* вказує саме на *a*.

Ці дві операції — **присвоєння адреси** та **розіменування** — є базовими, але критично важливими для розуміння роботи з вказівниками, функціями та динамічною пам’яттю.

1.2. Операції з вказівниками

У мові програмування С вказівники підтримують обмежену кількість арифметичних операцій, які визначаються специфікою їх використання для доступу до елементів масивів і роботи з пам’яттю. На відміну від звичайних числових змінних, вказівники при арифметичних операціях враховують розмір типу даних, на який вони вказують. Це дозволяє правильно переходити між елементами структурованих даних у пам’яті.

Дозволені арифметичні операції: додавання цілого числа до вказівника, віднімання цілого числа від вказівника, різниця між двома вказівниками, інкремент/декремент вказівника.

Операція $p + n$ зсуває вказівник p на n елементів уперед:

```
int arr [] = {10, 20, 30, 40};
int *ptr = arr;
printf ("%d\n", *(ptr + 2)); // виведе 30
```

тут $ptr + 2$ переходить на третій елемент масиву arr.

Операція $p - n$ зсуває вказівник назад на n елементів:

```
int arr [] = {5, 15, 25, 35};
int *ptr = &arr [3];
printf ("%d\n", *(ptr - 2)); // виведе 15
```

тут вказівник переміщується з останнього елемента на другий.

Можна визначити кількість елементів між двома вказівниками, які вказують на елементи одного масиву:

```
int arr [] = {1, 2, 3, 4, 5};
int *start = &arr [1];
int *end = &arr [4];
int diff = end – start;
printf ("%d\n", diff); // виведе 3
```

Ця операція повертає кількість елементів між $start$ і end.

Збільшення або зменшення вказівника на 1 переміщує його на один елемент відповідно вперед або назад:

```
int data [] = {100, 200, 300};
int *p = data;
p++; // вказує на data [1]
printf ("%d\n", *p); // виведе 200
p--; // знову вказує на data [0]
printf ("%d\n", *p); // виведе 100
```

Зауваження:

- всі арифметичні операції є коректними лише в межах одного масиву або ділянки пам’яті;
- вказівники не можна довільно додавати або множити один на одного;
- арифметика вказівників не гарантує безпеки доступу — програміст несе повну відповідальність за коректність адрес.

У мові С також дозволено виконувати **порівняння вказівників**, якщо вони вказують на елементи **одного масиву або ділянки пам’яті**. Порівняння виконується на основі адрес, які зберігають вказівники, тобто фактично порівнюються числові значення пам’яті.

Дозволені оператори порівняння:

- `= =` — рівність (true, якщо вказівники вказують на одну й ту саму адресу);
- `! =` — нерівність;
- `<`, `>`, `<=`, `>=` — порівняння відносного розташування в пам’яті.

Наприклад, перевірка рівності вказівників:

```
int a = 5, b = 5;
int *p1 = &a;
int *p2 = &b;
if (p1! = p2) {
   printf ("Вказівники різні.\n");
        }
```

Навіть якщо значення змінних однакові, p1 і p2 вказують на різні області пам’яті.

Наприклад, порівняння вказівників у межах масиву:

```
int arr [] = {10, 20, 30, 40};
int *p = &arr [1];
int *q = &arr [3];
if (p < q) {
   printf ("p вказує на елемент раніше в масиві, ніж q.\n");
        }
```

Оскільки p і q вказують на елементи одного масиву, порівняння є коректним.

Зауваження:

- поведінка є **невизначеною**, якщо порівнюються вказівники на **різні об'єкти** (наприклад, два різні масиви);
- дозволено порівнювати вказівник із NULL, щоб перевірити, чи він не ініціалізований.

```
int *ptr = NULL;
if (ptr == NULL) {
   printf ("Вказівник не вказує на жодну адресу.\n");
        }
```

Таким чином, порівняння вказівників — це потужний інструмент для контролю ітерацій у масивах, перевірки на неініціалізованість і організації умовних переходів. Проте воно повинно виконуватись лише в межах однієї області пам’яті, щоб уникнути непередбачуваної поведінки.

1.3. Динамічне виділення пам’яті

У мові С динамічне виділення пам’яті дає змогу програмі резервувати пам’ять під час виконання, що особливо корисно, коли розміри структур або масивів заздалегідь невідомі. Таке виділення здійснюється у спеціальній області пам’яті, яка називається купа (heap).

Купа (heap) — це ділянка оперативної пам’яті, керування якою здійснюється вручну програмістом через спеціальні функції бібліотеки $< stdlib.h >$. На відміну від стеку, де пам’ять звільняється автоматично після завершення функції, пам’ять у купі зберігається доти, доки вона явно не буде звільнена. Неправильне керування купою може призвести до витоків пам’яті або невизначеної поведінки програми.

У стандартній бібліотеці $< stdlib.h >$ передбачено низку функцій: $malloc(), calloc(), realloc()$ і $free()$.

Функція $malloc(size_t\ size)$ виділяє неперервний блок пам’яті розміром size байтів та повертає вказівник на її початок. Вміст цієї пам’яті не ініціалізується, тобто значення в ній будуть випадковими.

Приклад:

```
int *arr = (int *) malloc (5 * sizeof(int));
if (arr == NULL) {
   printf ("Помилка виділення пам’яті.\n");}
```

У цьому прикладі виділяється пам’ять під масив з п’яти елементів типу int. Значення елементів — невизначені, тому їх необхідно проініціалізувати перед використанням.

Функція $calloc(size_t\ num, size_t\ size)$ також виділяє блок пам’яті, але на відміну від $malloc()$, вона ініціалізує всі байти нулями. Аргументи — кількість елементів і розмір кожного.

Приклад:

```
int *arr = (int *) calloc (5, sizeof(int));
if (arr == NULL) {
   printf ("Помилка виділення пам’яті.\n");
       }
```

У цьому випадку створюється масив з п’яти цілих чисел, кожен з яких дорівнює нулю.

Функція $realloc(void\ *ptr, size_t\ new_size)$ змінює розмір раніше виділеного блоку пам’яті. Вона або розширює наявний блок, якщо це можливо, або створює новий блок пам’яті, копіює в нього наявні дані, звільняє старий. Якщо $ptr == NULL$, то $realloc()$ поводиться як $malloc$ (). Якщо $new_size == 0$, то $realloc()$ буде працювати як $free()$. Наприклад:

```
int *arr = (int *) malloc (3 * sizeof(int));
if (arr = = NULL) return 1;
arr [0] = 10; arr [1] = 20; arr [2] = 30;
int *temp = realloc (arr, 6 * sizeof(int));
if (temp ! = NULL) {
   arr = temp;
   arr [3] = 40;     arr [4] = 50;     arr[5] = 60; }
```

Зверніть увагу: завжди слід використовувати проміжний вказівник ($temp$), щоб уникнути втрати доступу до початкового блоку в разі помилки виділення.

Функція $free(void\ *ptr)$ звільняє пам'ять, раніше виділену через $malloc()$, $calloc()$ або $realloc()$. Після виклику $free()$ доступ до цієї пам'яті заборонений. Наприклад:

```
free(arr);
arr = NULL; // рекомендується обнулити вказівник
```

Обнулення вказівника після $free()$ запобігає випадковому доступу до вже звільненої пам'яті (dangling pointer).

Порівняння $malloc()$, $calloc()$ та $realloc()$:

Функція	Ініціалізація	Можливість зміни розміру	Призначення
$malloc()$	ні	ні	Виділення пам'яті без ініціалізації
$calloc()$	так (0)	ні	Виділення пам'яті з ініціалізацією нулями
$realloc()$	частково	так	Зміна розміру існуючого блоку пам'яті

Функції $malloc()$, $calloc()$, $realloc()$ і $free()$ утворюють основу для ручного керування пам'яттю. Їх коректне використання дає змогу створювати ефективні, масштабовані програми. Водночас це вимагає особливої уважності: будь-яке недбале ставлення до звільнення пам'яті або перевірки результатів може призвести до серйозних помилок — витоків пам'яті або аварійного завершення програми.

1.4. Одновимірні динамічні масиви

У мові програмування С одновимірні масиви зазвичай створюються статично з фіксованою кількістю елементів. Проте в багатьох задачах потрібно створювати масиви, розмір яких невідомий під час компіляції або може змінюватися під час виконання програми. Для цього використовуються **одновимірні динамічні масиви**, що створюються за допомогою механізмів **динамічного виділення пам'яті**.

Для створення одновимірного динамічного масиву застосовують функцію $malloc()$ або $calloc()$. Результатом є **вказівник на перший елемент масиву**, що далі використовується як звичайний масив. Приклад виділення пам'яті на одновимірний масив цілих чисел за допомогою функції $malloc()$:

```
int *A = (int *) malloc (n * sizeof(int));
```

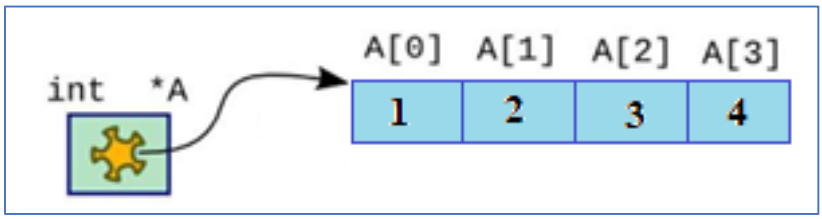


Рис.1.2. Динамічне виділення пам'яті на одновимірний масив, $n = 4$

Приклад виділення пам'яті на одновимірний масив цілих чисел за допомогою функції $calloc()$:

```
int *arr = (int *) calloc (n, sizeof(int));
```

У обох випадках n — кількість елементів у масиві, $sizeof(int)$ - розмір одного елемента масиву, результат $malloc()/calloc()$ приведено до типу $int*$.

Доступ до елементів динамічного масиву здійснюється так само, як у звичайному масиві — за індексом:

```
arr [0] = 5;
arr [1] = 10;
```

альтернативно — через арифметику вказівників:

```
*(arr + 2) = 15;
```

Приклад виділення, використання і звільнення пам'яті

```
#include <stdio.h>
#include <stdlib.h>
int main () {
   int n;
   printf ("Введіть розмір масиву: ");
   scanf ("%d", &n);
   int *arr = (int *) malloc (n * sizeof(int));
   if (arr = = NULL) {
      printf ("Помилка виділення пам’яті.\n");
      return 1;
   }
   // Заповнення масиву
   for (int i = 0; i < n; i++) {
      arr[i] = i * i; // або, використовуючи адреси:   *(arr + i) = i * i;
   }
   // Виведення масиву
   printf ("Елементи масиву: \n");
   for (int i = 0; i < n; i++) {
      printf ("%d ", arr[i]); // або адресно:  printf ("%d ", *(arr + i));
   }
   // Звільнення пам’яті
   free(arr);
   return 0; }
```

Переваги динамічних масивів:

- гнучкість (розмір масиву задається під час виконання),
- ефективність використання пам’яті (пам’ять виділяється лише за потреби),
- можливість масштабування (розмір масиву можна змінити через $realloc()$).

Якщо розмір масиву потрібно змінити під час виконання, використовується функція $realloc()$:

```
int * arr;
arr = (int *) realloc (arr, new_size * sizeof(int))
```

Функція $realloc()$ дозволяє створювати **масиви, розмір яких може збільшуватись (зменшуватись)**, наприклад, при введенні даних користувачем без обмежень.

Приклад. Створити програму, яка дозволяє користувачеві вводити довільну кількість цілих чисел. Введення завершується, коли користувач вводить -1. Програма зберігає всі введені значення у динамічному масиві, а потім обчислює та виводить середнє значення всіх введених чисел.

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    int *arr = NULL;
    int size = 0;
    int capacity = 2;
```

```
        int input;
        arr = (int *) malloc (capacity * sizeof(int));
        if (arr == NULL) {
            printf ("Помилка виділення пам’яті.\n");
    return 1;}
printf ("Введіть цілі числа (для завершення введіть -1): \n");
  while (1) {
    scanf ("%d", &input);
    if (input = = -1) break;
    // Якщо масив заповнений — розширюємо
    if (size == capacity) {
      capacity *= 2;
      int *temp = realloc (arr, capacity * sizeof(int));
      if (temp == NULL) {
        printf ("Помилка при розширенні пам’яті.\n");
        free(arr);
        return 1;
      }
      arr = temp;
    }
    arr[size++] = input;
  }
  if (size == 0) {
    printf ("Не введено жодного числа.\n");
  } else {
    int sum = 0;
    for (int i = 0; i < size; i++) {
      sum += arr[i];
    }
```

```
    double average = (double)sum / size;
    printf ("Середнє значення: %.2f\n", average);
}
free(arr);
return 0;}
```

Одновимірні динамічні масиви дозволяють реалізовувати ефективні структури даних з довільним обсягом, але водночас вимагають від програміста точності та уважності при роботі з пам’яттю. Їхнє правильне використання є запорукою ефективного керування ресурсами програми.

1.5. Двовимірні динамічні масиви

У мові програмування С важливою особливістю є можливість динамічного керування пам’яттю, що дозволяє створювати структури даних змінного розміру, зокрема двовимірні масиви. Такі масиви використовуються для зберігання табличних даних, матриць коефіцієнтів, елементів сіток тощо. На відміну від статичних масивів, розмір яких задається під час компіляції, динамічні масиви дозволяють адаптувати структуру до потреб програми вже під час виконання.

Двовимірний масив у С реалізується як масив вказівників, кожен з яких посилається на окремий одновимірний масив. Це дозволяє гнучко змінювати кількість рядків і стовпців, а також створювати нерівномірні масиви (jagged arrays), у яких кожен рядок має власну довжину. Для виділення пам’яті застосовуються функції стандартної бібліотеки $< stdlib.h >$, зокрема $malloc()$.

Коли розміри масиву є відомими під час виконання програми, спочатку виділяється пам’ять під масив вказівників на рядки. Після цього для кожного з цих вказівників виділяється пам’ять під відповідний рядок значень. Звернення до елементів здійснюється через подвійні індекси — у формі matrix[i][j], що відповідає стандартному синтаксису доступу до елементів масиву в C.

Нижче наведено приклад створення двовимірного динамічного масиву цілих чисел розміром 3×4, його заповнення вручну, виведення на екран, а також демонстрація двох підходів до звернення до елементів: через індекси та через арифметику вказівників.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int rows = 3;
    int cols = 4;
    // Виділення пам’яті для масиву вказівників
    int **matrix = (int **)malloc(rows * sizeof(int *));
    if (matrix == NULL) {
        printf("Помилка виділення пам’яті.\n");
        return 1;
    }
    // Виділення пам’яті для кожного рядка
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int *)malloc(cols * sizeof(int));
        if (matrix[i] == NULL) {
            printf("Помилка виділення пам’яті для рядка %d.\n", i);
```

```
            // Звільнення вже виділеної пам’яті
            for (int k = 0; k < i; k++) {
                free(matrix[k]);
            }
            free(matrix);
            return 1;
        }
    }
    // Заповнення масиву: приклад з множенням індексів
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = i * j;
        }
    }
    // Виведення елементів через індекси
    printf("Вивід масиву через індекси:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
    // Виведення елементів через арифметику вказівників
    printf("Вивід масиву через адреси:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", *(*(matrix + i) + j));
        }
```

```
    printf("\n");
}
// Звільнення пам’яті
for (int i = 0; i < rows; i++) {
    free(matrix[i]);
}
free(matrix);
return 0;
    }
```

У цьому прикладі застосовано дві форми доступу до елементів масиву: стандартну форму індексації та доступ через арифметику вказівників, що демонструє взаємозамінність цих підходів у мові С. Така гнучкість є потужною рисою мови, однак вимагає від програміста точного розуміння адресної арифметики та моделі пам’яті (див. Рис.1.2).

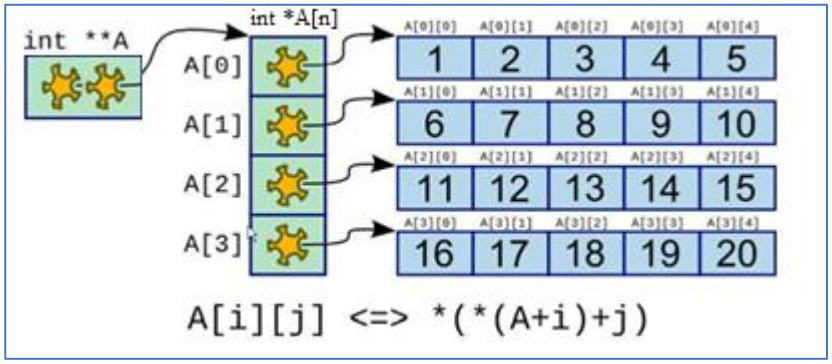


Рис.1.3. Динамічне виділення пам’яті на двовимірний масив

Зазначимо також, що при роботі з динамічною пам’яттю важливо забезпечити належне її звільнення після завершення використання масиву, інакше відбувається витік пам’яті — проблема, що негативно впливає на стабільність та продуктивність програм.

Окрім числових масивів, аналогічна методика застосовується для символьних двовимірних масивів, наприклад, при зберіганні масиву

рядків. У цьому випадку кожен рядок може містити слово або текстовий фрагмент, і знову ж таки застосовується масив вказівників типу $char **$, де кожен елемент вказує на окремий символьний масив, що завершується нульовим символом ('\0').

Як і у випадку з числовими масивами, двовимірний символьний масив може бути реалізований за допомогою динамічного виділення пам’яті. Зазвичай такий масив застосовується для зберігання набору рядків, наприклад, списку слів чи текстових рядків. У цьому випадку кожен рядок масиву є окремим символьним масивом, а загальний масив — це масив вказівників на ці рядки.

Для реалізації двовимірного масиву рядків виділяється пам’ять для масиву вказівників на char*, а потім для кожного вказівника виділяється пам’ять для окремого рядка символів, що завершується нульовим символом ('\0').

Приклад:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
   int rows = 3;
   int cols = 20; // Максимальна довжина кожного рядка
   // Виділення пам’яті під масив вказівників
   char **matrix = (char **)malloc(rows * sizeof(char *));
   if (matrix == NULL) {
      printf("Помилка виділення пам’яті для рядків.\n");
      return 1;
   }
```

```
// Виділення пам’яті для кожного рядка
for (int i = 0; i < rows; i++) {
    matrix[i] = (char *)malloc(cols * sizeof(char));
    if (matrix[i] == NULL) {
        printf("Помилка виділення пам’яті для рядка %d.\n", i);
        // Звільнення вже виділеної пам’яті
        for (int k = 0; k < i; k++) {
            free(matrix[k]);
        }
        free(matrix);
        return 1;
    }
}
// Заповнення масиву рядками
strcpy(matrix[0], "Привіт");
strcpy(matrix[1], "Світ");
strcpy(matrix[2], "Мова C");
// Виведення елементів через індекси
printf("Вивід масиву рядків через індекси:\n");
for (int i = 0; i < rows; i++) {
    printf("%s\n", matrix[i]);
}
// Виведення елементів через арифметику вказівників
printf("Вивід масиву рядків через адреси:\n");
for (int i = 0; i < rows; i++) {
    printf("%s\n", *(matrix + i));
}
// Звільнення пам’яті
for (int i = 0; i < rows; i++) {
```

```
    free(matrix[i]);
}
free(matrix);
return 0;
    }
```

У цьому прикладі створюється двовимірний масив для зберігання трьох рядків, кожен з яких може мати до 20 символів. Мова йде про динамічне виділення пам’яті під масив вказівників типу char*, де кожен елемент масиву є вказівником на масив символів, що представляє рядок. Кожен рядок у цьому випадку містить текст, збережений за допомогою стандартної бібліотечної функції strcpy(), що копіює рядок в область пам’яті.

Звернення до елементів масиву також можна здійснювати двома способами: через індекси (matrix[i]) або за допомогою арифметики вказівників (*(matrix + i)), що демонструє гнучкість доступу до елементів.

Після використання пам’яті вона звільняється за допомогою функції free(), що є обов’язковим кроком для запобігання витокам пам’яті.

Використання динамічних двовимірних масивів дає можливість ефективно працювати з табличними даними змінного розміру, такими як списки рядків або таблиці. Цей підхід дозволяє зберігати дані з адаптивним розміром, при цьому забезпечуючи гнучкість при доступі до елементів через індексацію чи арифметику вказівників. Ключовим аспектом є коректне керування пам’яттю, що включає її виділення, використання та звільнення після завершення роботи з масивами.

Питання для самоконтролю

1. Що таке вказівник в мові С? Як він відрізняється від звичайної змінної?
2. Які основні типи даних можуть бути вказівниками? Наведіть приклади.
3. Як оголосити вказівник на змінну типу int?
4. Що таке адреса змінної і як її можна отримати за допомогою оператора &?
5. Як отримати значення змінної за допомогою вказівника? Поясніть, що таке розіменування.
6. Як можна здійснити арифметичні операції з вказівниками? Наведіть приклад інкременту вказівника.
7. Що таке арифметика вказівників? Як змінити адресу, на яку вказує вказівник?
8. Поясніть різницю між оператором * в контексті оголошення вказівника та в контексті розіменування вказівника.
10. Які наслідки можуть бути від неправильного використання арифметики вказівників?
11. Що таке динамічне виділення пам’яті? Які функції мови С використовуються для цього?
12. У чому різниця між функціями malloc() і calloc()?
13. Як використовувати функцію malloc() для виділення пам’яті під масив з 10 елементів типу int?
14. Як можна змінити розмір уже виділеного блоку пам’яті за допомогою функції realloc()? Наведіть приклад.

15. Що відбувається, якщо ви не звільняєте пам’ять, виділену за допомогою malloc(), calloc() чи realloc()?
16. Як правильно звільняти пам’ять після використання функції динамічного виділення?
17. Як оголосити одновимірний динамічний масив у C? Напишіть код для виділення пам’яті під масив на 20 елементів типу float.
18. Як заповнити одновимірний динамічний масив значеннями, зчитаними з клавіатури?
19. Як правильно звільнити пам’ять виділену для одновимірного динамічного масиву після використання?
20. Які переваги та недоліки має використання динамічних масивів у порівнянні зі статичними?
21. Як оголосити двовимірний динамічний масив у C? Напишіть код для створення матриці розміру 3x3.
22. Як виділити пам’ять для двовимірного масиву за допомогою malloc() або calloc()? Покажіть приклад.
23. Як звертатися до елементів двовимірного динамічного масиву? Наведіть приклад доступу до елемента.
24. Як змінити розмір двовимірного динамічного масиву за допомогою realloc()? Напишіть код для цього.
25. Як правильно звільняти пам'ять виділену для двовимірних динамічних масивів?

Практичні завдання

Завдання 1.

Розробити програму, яка виконує операції над елементами одномірного масиву цілих чисел $A[n]$, використовуючи вказівники. Пам'ять для масиву має виділятися динамічно. Реалізувати доступ до елементів масиву за допомогою вказівників (через адреси) або індексним способом.

1. Знайти суму та добуток всіх елементів масиву.
2. Визначити кількість парних елементів у масиві.
3. Визначити середнє арифметичне всіх додатних елементів масиву.
4. Визначити мінімальний елемент серед непарних чисел.
5. Знайти індекс останнього максимального елемента масиву.
6. Знайти різницю між максимальним і мінімальним елементами масиву.
7. Підрахувати кількість елементів, менших за середнє арифметичне масиву.
8. Знайти перший парний елемент масиву та його індекс.
9. Визначити добуток всіх від'ємних елементів масиву.
10. Визначити, чи є серед елементів масиву хоча б один нуль.
11. Знайти найближчий до нуля елемент масиву.
12. Знайти найбільший серед від'ємних елементів масиву.
13. Замінити всі від'ємні елементи масиву їхніми модулями.
14. Обчислити різницю між сумою парних і сумою непарних елементів масиву.
15. Визначити, чи є масив симетричним відносно його середини.
16. Визначити, чи впорядкований масив за зростанням.

Завдання 2.

Розробити програму, яка виконує операції над елементами одномірного масиву цілих чисел $A[n]$, використовуючи вказівники. Пам'ять для масиву має виділятися динамічно. Реалізувати доступ до елементів масиву за допомогою вказівників (через адреси).

1. Створити новий масив $B[k]$, з тих елементів масиву $A[n]$, які більші за середнє арифметичне масиву $A[n]$.
2. Замінити всі від'ємні елементи динамічного масиву на їхні модулі та помістити їх на початок масиву.
3. Переставити всі парні елементи на початок масиву, а непарні – в кінець.
4. Перемістити всі нульові елементи масиву в кінець, зберігши порядок інших елементів.
5. Видалити всі нульові елементи з масиву та змінити його розмір.
6. Виконати реверс динамічного масиву, використовуючи лише вказівники.
7. Створити новий масив розміру **2N**, скопіювати в нього початковий масив та його обернену копію.
8. Виділити пам'ять для нового масиву, що містить тільки непарні елементи початкового масиву.
9. Обчислити кількість елементів, що є квадратами цілих чисел.
10. Виконати циклічний зсув елементів динамічного масиву праворуч на **k** позицій.
11. Виконати циклічний зсув елементів динамічного масиву ліворуч на **k** позицій.

12. Виділити новий масив, що містить тільки унікальні значення початкового масиву.
13. Виконати перетин двох динамічних масивів і зберегти результат у новому масиві.
14. Об’єднати два динамічні масиви, впорядкувавши їх за зростанням.
15. Розробіть програму, яка для кожного елемента масиву цілих чисел замінює його на суму цифр цього числа.

    Приклад:

    Вхідний масив: {23, 7, 104}. Результат: {5, 7, 5}
16. Напишіть програму, яка переміщує всі нулі в кінець масиву, зберігаючи при цьому порядок інших елементів. Не використовуйте додаткових масивів для реалізації цього завдання.

    Приклад:

    Вхідний масив: {4, 0, 2, 0, 5, 7, 0}. Результат: {4, 2, 5, 7, 0, 0, 0}.
17. Розробіть програму, яка видаляє всі елементи масиву, що діляться на задане число k. Розмір масиву після видалення елементів має коректно оновлюватися за допомогою динамічного виділення пам'яті (за допомогою realloc()).

    Приклад:

    Вхідний масив: {10, 3, 5, 15, 8}, k = 5. Результат: {3, 8}.

Завдання 3.

Розробити програму, яка виконує операції над елементами двовимірного масиву цілих чисел $A[n][m]$, використовуючи вказівники. Пам’ять для масиву має виділятися динамічно. Реалізувати доступ до

елементів масиву за допомогою вказівників (через адреси) або індексним способом.

1. Визначити суму всіх елементів двовимірного масиву.
2. Знайти максимальний елемент у кожному рядку масиву.
3. Визначити середнє арифметичне кожного стовпця масиву.
4. Підрахувати кількість від'ємних елементів у масиві.
5. Визначити кількість парних чисел у масиві.
6. Визначити мінімальний елемент у кожному стовпці масиву.
7. Переставити місцями перший та останній рядки масиву.
8. Переставити місцями перший та останній стовпці масиву.
9. Виконати транспонування квадратного масиву.
10. Визначити, чи є масив симетричним відносно головної діагоналі.
11. Виконати циклічний зсув рядків масиву вниз на одну позицію.
12. Виконати циклічний зсув стовпців масиву вправо на одну позицію.
13. Обчислити суму елементів, що розташовані вище головної діагоналі.
14. Знайти максимальний елемент у кожному рядку та записати його у одновимірний масив $B[n]$.
15. Визначити добуток елементів побічної діагоналі масиву $A[n][n]$.
16. Визначити мінімальний елемент кожного рядка та переставити рядки у порядку зростання цих мінімальних значень.
17. Обчислити середнє арифметичне кожного стовпця та впорядкувати стовпці масиву за спаданням їхніх середніх арифметичних значень.

18. Перетворити масив у трикутну матрицю, занулюючи елементи під головною діагоналлю.
19. Виконати циклічний зсув рядків масиву вгору на k позицій.
20. Виконати циклічний зсув стовпців масиву вліво на k позицій.