МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт
з дисципліни «Програмування»
для студентів денної та заочної форм навчання
за спеціальністю 123 «Комп’ютерна інженерія»

Частина 1

Затверджено
редакційно-видавничою
радою університету,
протокол № 3 від 06.10.2021 р.

Харків
НТУ «ХПІ»
2021

Методичні вказівки до виконання лабораторних робіт з дисципліни «Програмування» для студентів денної та заочної форм навчання за спеціальністю 123 «Комп'ютерна інженерія». Ч. 1 / уклад. : Порошин С. М., Статкус А. В., Носик А. М., Онищенко В. В. – Харків: НТУ «ХПІ», 2021. – 158 с.

Укладачі: С. М. Порошин
А. В. Статкус
А. М. Носик
В. В. Онищенко

Рецензент В. В. Усик

Кафедра мультимедійних інформаційних технологій і систем

ЗМІСТ

Вступ …… 4

Лабораторна робота №1

Дослідження основних можливостей інтегрованого середовища розробки *Code::Blocks* для роботи з *C*-програмами …… 5

Лабораторна робота №2

Дослідження основних принципів алгоритмізації обчислювальних процесів ……..... 18

Лабораторна робота №3

Дослідження базових типів даних мови *C* ……... 27

Лабораторна робота №4

Дослідження інструкцій вибору …… 58

Лабораторна робота №5

Дослідження методів сортування одновимірного масиву …….. 70

Лабораторна робота №6

Дослідження способів передачі аргументів функцій користувача …… 90

Лабораторна робота №7

Дослідження принципів застосування вказівників …….. 105

Лабораторна робота №8

Дослідження основних можливостей роботи з файлами …… 115

Список джерел інформації ……. 137

Додатки …….. 138

Додаток 1 Гарячі клавіші *Code::Blocks* …… 138

Додаток 2 Результати роботи для завдань 4 і 5 лабораторної роботи №5 … 141

Додаток 3 Формули для визначення площ геометричних фігур …… 143

Додаток 4 Варіанти завдання 3 лабораторної роботи №6 …….. 148

Додаток 5 Приклад програми для роботи з функціями …….. 156

ВСТУП

Програмування має дуже багато сфер застосування, де інформаційні технології (ІТ) набувають розвитку дуже швидкими темпами. Розробники програмних продуктів потрібні в багатьох сферах, які навіть не завжди пов'язані тільки з використанням ІТ.

Метою вивчення навчальної дисципліни «Програмування» є отримання навичок побудови основних обчислювальних алгоритмів, вивчення основ мови програмування високого рівня *С* та розробки комп'ютерних програм, ознайомлення з методами розв'язання обчислювальних задач та формування вмінь з їх практичного використання.

Багато фірм, що розробляють програмне забезпечення, все частіше звертаються до *С* як до зручної мови для реалізації своїх проєктів, оскільки відомо, що *С* дозволяє отримати компактні та ефективні програми. Мобільність і продуктивність мови *С* є одними з найбільш затребуваних її характеристик при реалізації таких операційних систем як Linux, Microsoft Windows, а також Google Android. Мова *С* є однією з найпопулярніших мов для розробки вбудованих систем, від яких зазвичай вимагається висока швидкість виконання та низьке витрачання пам'яті, а також для систем реального часу та зв'язку.

Дисципліна «Програмування» є базовою для вивчення інших дисциплін, пов'язаних з питаннями алгоритмізації та програмування.

Успіх у вивченні тієї чи іншої мови програмування неможливий без закріплення отриманих знань під час виконання лабораторних робіт. Представлені методичні вказівки до виконання лабораторних робіт з дисципліни «Програмування» призначені для опрацювання питань, пов'язаних з роботою в інтегрованому середовищі розробки *Code::Blocks*, розглядом основних принципів алгоритмізації обчислювальних процесів, дослідженням базових типів мови *С*, роботою з інструкціями вибору, масивами, стандартними функціями та функціями користувача, вказівниками, а також роботою з файлами.

ЛАБОРАТОРНА РОБОТА №1
ДОСЛІДЖЕННЯ ОСНОВНИХ МОЖЛИВОСТЕЙ ІНТЕГРОВАНОГО СЕРЕДОВИЩА РОЗРОБКИ CODE::BLOCKS ДЛЯ РОБОТИ З *C*-ПРОГРАМАМИ

Мета роботи: ознайомитися з зовнішнім виглядом та основними можливостями інтегрованого середовища розробки *Code::Blocks*; навчитися створювати консольні застосунки, редагувати та запускати на виконання прості *C*-програми; засвоїти роботу з пошуку та усунення помилок компілятора; навчитися відстежувати значення змінних у процесі виконання програми; розглянути можливість виводу на консоль україномовного тексту.

Теоретична частина

1.1. Інтегроване середовище розробки *Code::Blocks*

Інтегроване середовище розробки (IDE, Integrated Development Environment) *Code::Blocks* являє собою повноцінний продукт для розробки застосунків на мові C/C++ [1]. Встановити *Code::Blocks* можна за посиланням [2].

Code::Blocks – це багатоплатформне середовище, яке розроблене з використанням бібліотеки інструментів wxWidgets. Існують версії середовища як для Windows, так і для інших операційних систем (ОС) – Linux і Mac OS.

IDE *Code::Blocks* є абсолютно безкоштовним, має відкритий початковий код і побудоване повністю з використанням відкритих технологій. Однією з найсильніших рис *Code::Blocks* є підтримка створення застосунків wxWidgets – програмного забезпечення (ПЗ), яке для роботи з графічним інтерфейсом користувача (GUI, Graphical User Interface) використовує прикладний програмний інтерфейс API (Application Programming Interface) ОС.

Зовнішній вигляд вікна IDE *Code::Blocks* наведено на рис. 1.1.

Code::Blocks вміє працювати з великою кількістю компіляторів. Якщо в системі їх встановлено декілька – можна з легкістю перемикатися між ними, в

залежності від потреби. Крім того, IDE саме на етапі інсталяції просканує систему на наявність доступних компіляторів і запропонує задіяти один з них в якості основного. Існує можливість пізніше змінювати ці налаштування як глобально, так і окремо для проєктів.

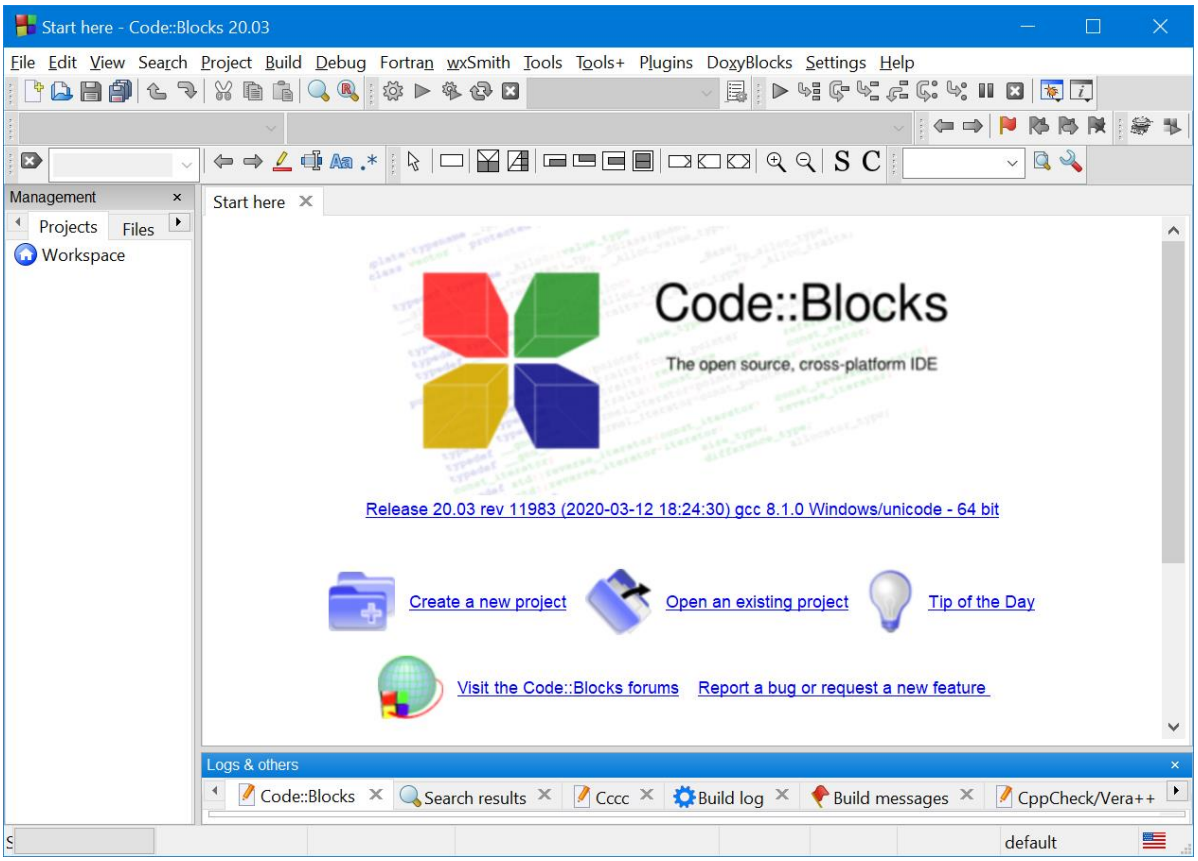


Рисунок 1.1 – Зовнішній вигляд вікна IDE Code::Blocks

Для загальних параметрів середовища існує цілий розділ «Settings» у головному меню (рис. 1.2). Щоб зробити будь-які специфічні налаштування, які є характерними для конкретного проєкту, потрібно відкрити вікно властивостей проєкту. Для цього необхідно викликати контекстне меню проєкту, натиснувши на ньому правою клавішею миші, і вибрати пункт «Properties». Самі проєкти (відкриті на даний момент в IDE) виведені у вигляді дерева в док-панелі «Management» в її вкладці «Projects».

У вікні властивостей можна налаштовувати різні параметри, починаючи від редагування цільового файлу і закінчуючи підключенням сторонніх біб-

ліотек. Кнопка «Project's build options» на вкладці «Project Settings» того ж самого вікна властивостей дозволить дуже гнучко налаштувати для конкретного проєкту конфігурацію збірки: ключі компілятора, вибрати сам компілятор, підключити бібліотеки для лінкування та інше. Все це можна виконати для окремих режимів «Release» і «Debug».

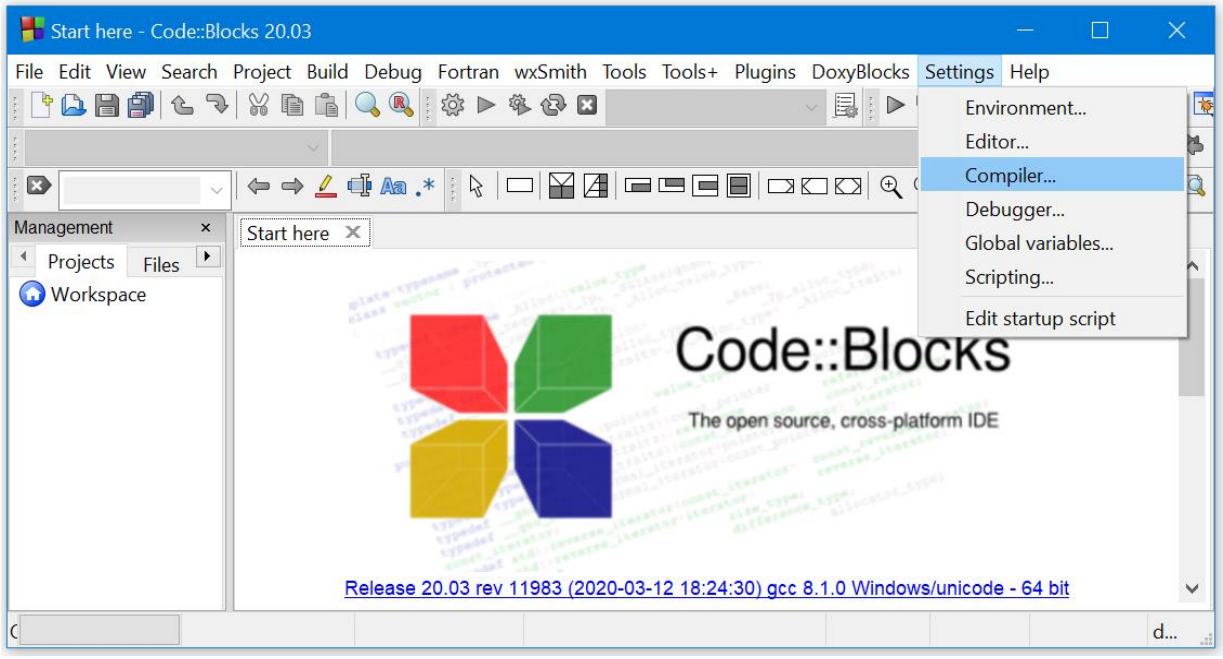


Рисунок 1.2 – Розділ «Settings» головного меню програми для налаштування загальних параметрів середовища

Графічний інтерфейс користувача *Code::Blocks* побудований на основі рухомих док-панелей, розмір яких може змінюватися. Крім того, їх можна пристиковувати до будь-яких сторін головного вікна, перетягуючи їх мишею. Завдяки цьому, можна дуже гнучко налаштовувати різні компонування інтерфейсу для наявних розмірів екрану, оптимізуючи ергономіку GUI.

Підсвічування, автодоповнення коду та засоби редагування в *Code::Blocks* роблять процес роботи з кодом легким і комфортним. Величезний арсенал налаштувань дозволяє дуже тонко пристосовувати середовище під конкретного користувача. Дуже корисним є наявність підтримки роботи зі змінним оточенням. Завдяки цьому середовище *Code::Blocks* можна дуже тісно інтегру-

вати до самої системи та розширювати його можливості. Наприклад, таким чином можна підключати бібліотеку wxWidgets.

Для розробки застосунків з GUI на основі wxWidgets в *Code::Blocks* є вбудований RAD-інструмент (Rapid Application Development) для конструювання GUI – wxSmith.

IDE *Code::Blocks* є дуже зручним середовищем для розробників на C/C++, а безкоштовне використання та багатоплатформна підтримка всіх сучасних ОС робить його одним з популярних IDE у своєму сегменті, поступаючись за потужністю мабуть лише Visual Studio.

Функціонал для стандарту C17 в *Code::Blocks* 20.03 за замовчуванням може бути відключений. Щоб його використовувати, потрібно обрати пункт меню Settings->Compiler.

У вкладці «Compiler Settings» (рис. 1.3) слід поставити галочки біля пунктів «Have gcc follow the 2017 ISO C language standard [-std=c17]», «Target x86_64 (64bit) [-m64]», «Produce debugging symbol [-g]» (для можливості проведення відлагодження програми) і натиснути кнопку «OK».

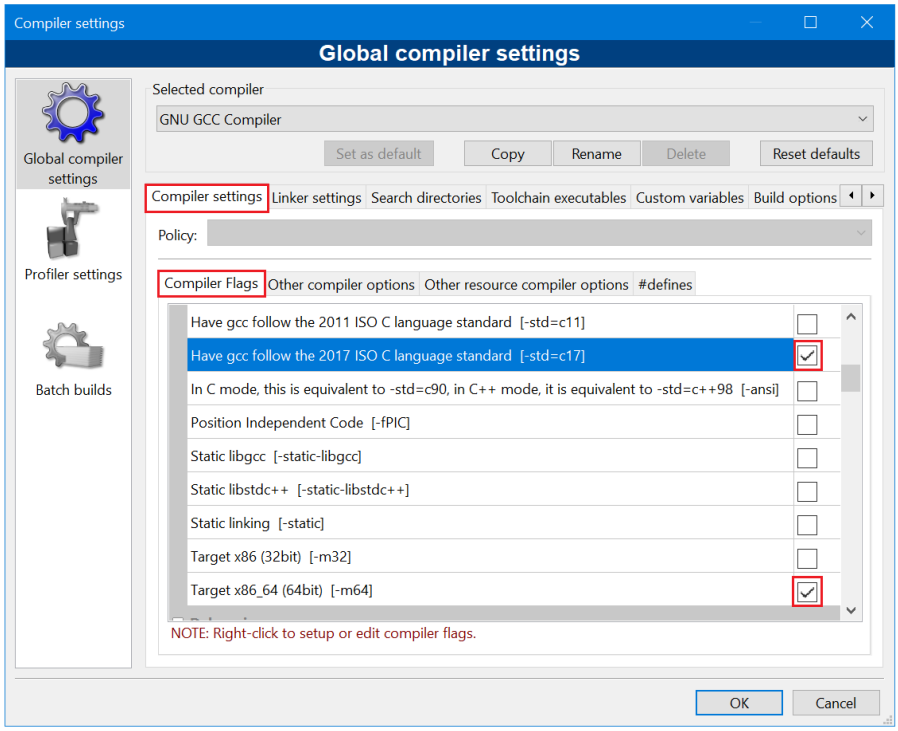


Рисунок 1.3 – Підключення можливостей для роботи зі стандартом C17

Крім того, слід перевірити правильність шляху до місця розташування компілятора (рис. 1.4) у вкладці «Toolchain executables». Наприкінці шляху повинен бути надпис «MinGW».

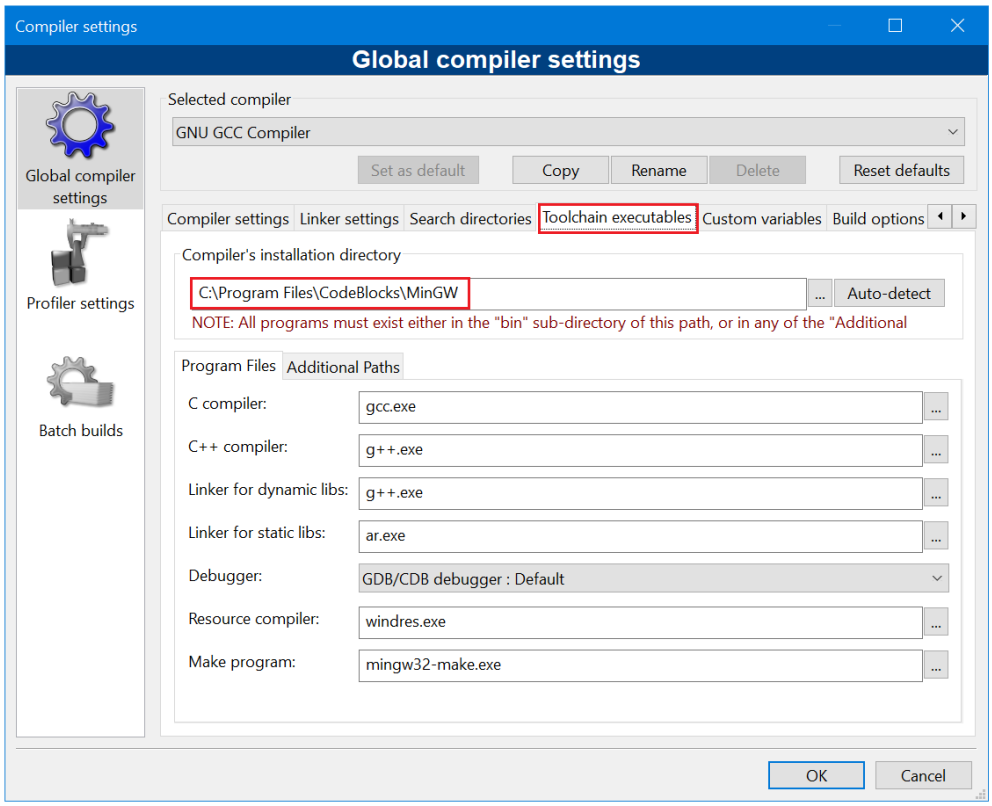


Рисунок 1.4 – Перевірка правильності шляху до компілятора

1.2. Створення проєкту для консольного застосунку

Якщо жодного проєкту в робочому просторі не створено, то після запуску *Code::Blocks* в центральній панелі активною буде лише вкладка «Start here». Для створення нового проєкту потрібно вибрати команду «Create a new project» або пункт меню «File->New->Project». У будь-якому випадку відкриється вікно вибору шаблону майбутнього проєкту, в якому необхідно вказати шаблон «Console application» і натиснути кнопку «Go» (рис. 1.5).

У вікні, що відкриється (рис. 1.6), необхідно поставити галочку для пункту «Skip this page next time» для того, щоб не показувати це вікно у подальшому. Після цього треба натиснути кнопку «Next». У наступному вікні (рис. 1.7) необхідно обрати мову *C* та натиснути кнопку «Next».

Далі слід вказати інформацію про ім’я майбутнього проєкту та місце, де він буде розташований на диску (рис. 1.8), і натиснути кнопку «Next».

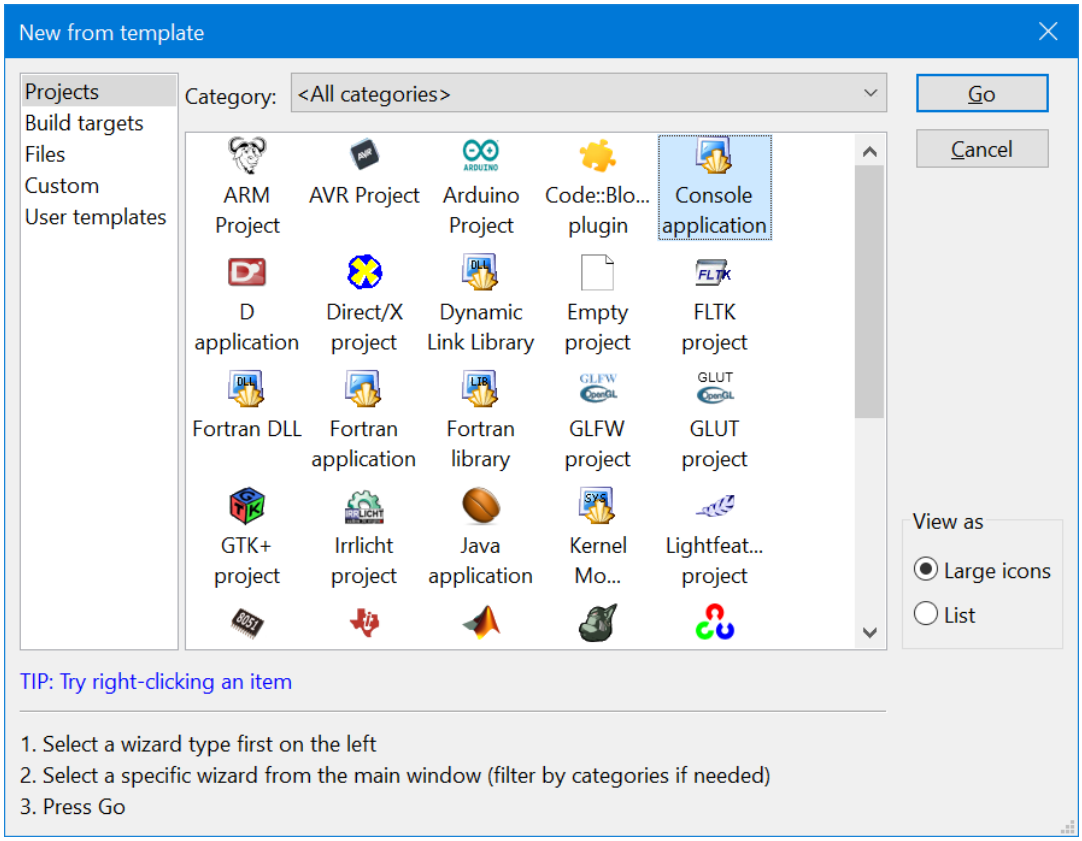


Рисунок 1.5 – Створення консольного застосунку

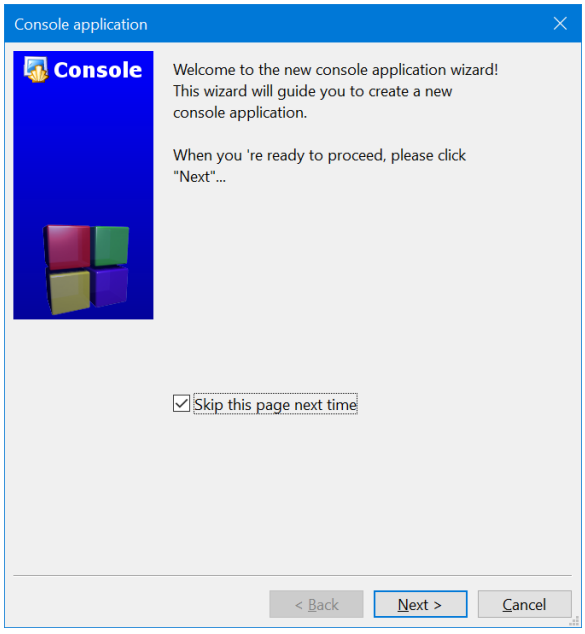


Рисунок 1.6 – Вікно, де слід поставити галочку, щоб воно більше не з’являлося

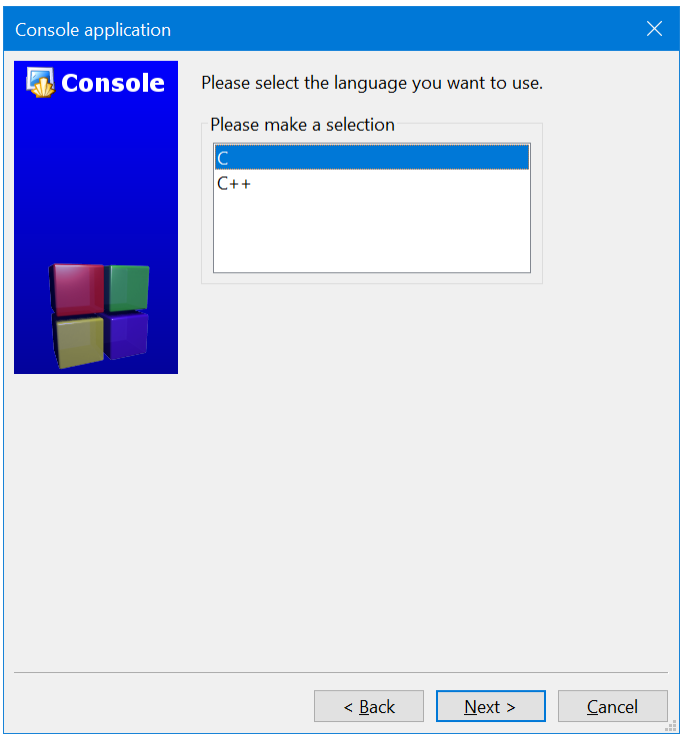


Рисунок 1.7 – Вікно для вибору мови програмування *C*

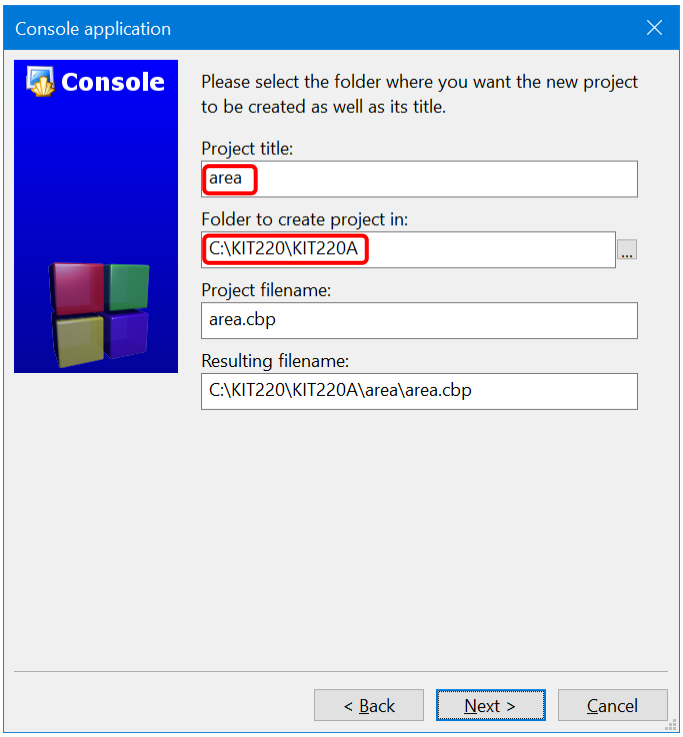


Рисунок 1.8 – Вікно для зазначення імені проєкту та місця його розташування

У вікні, що відкриється (рис. 1.9), необхідно натиснути кнопку «Finish» для завершення створення проєкту.

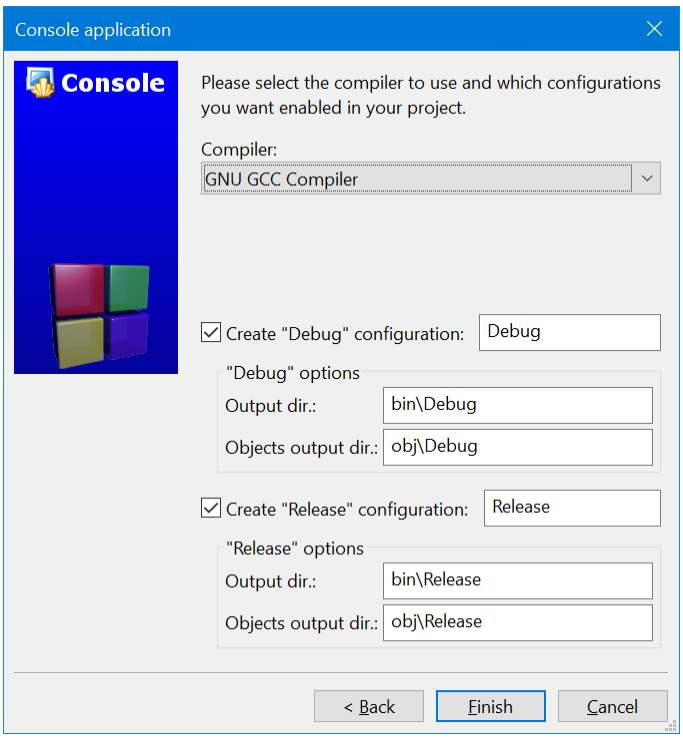


Рисунок 1.9 – Вікно для завершення процесу створення нового проєкту

Далі в док-панелі «Management» в її вкладці «Projects» необхідно двічі натиснути на **`main.c`** (рис. 1.10).

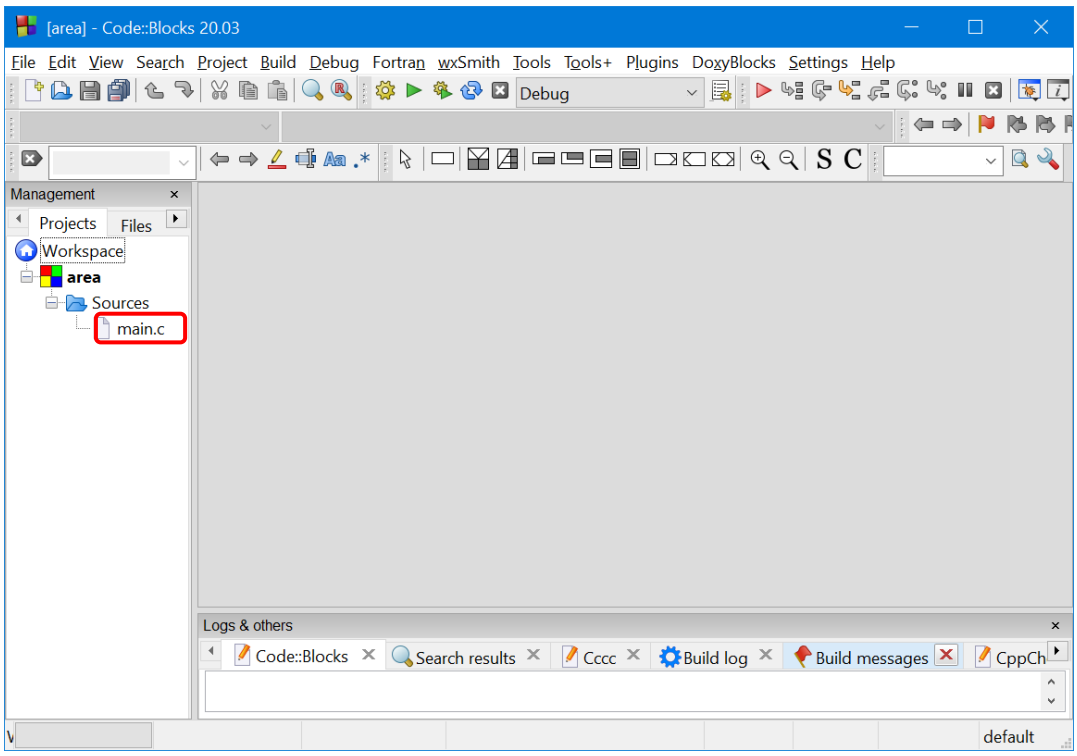


Рисунок 1.10 – Вікно, що відкривається після створення нового проєкту

В результаті праворуч у вікні (рис. 1.11) відкриється шаблон з готовою *C*-програмою, яку можна буде змінювати за власним бажанням.

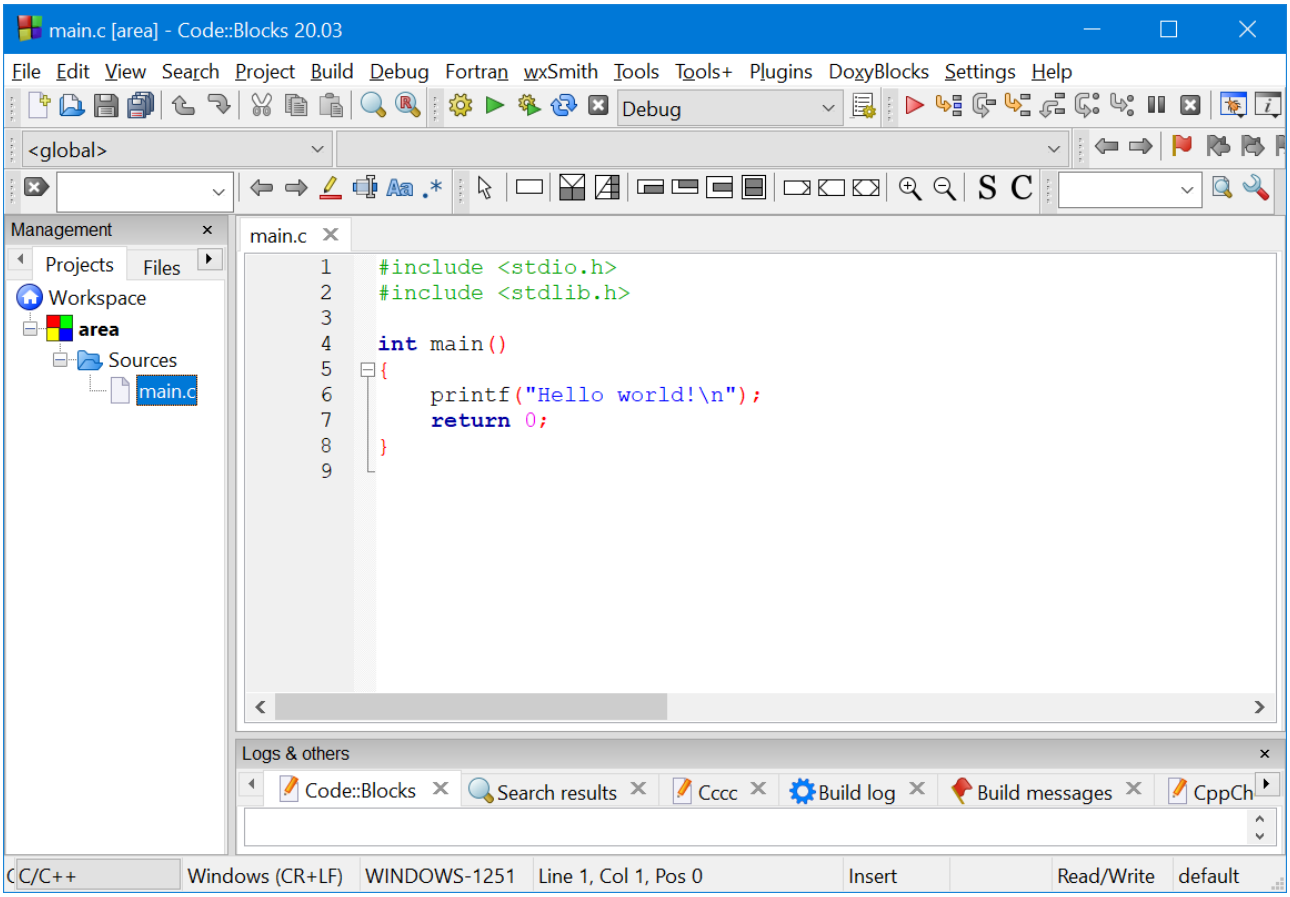


Рисунок 1.11 – Шаблон з готовою *C*-програмою

1.3. Робота в інтегрованому середовищі Code::Blocks

Напишемо текст програми для обчислення площі трикутника (рис. 1.12) за формулою Герона та введемо цей текст у вікні **`main.c`** замість шаблону.

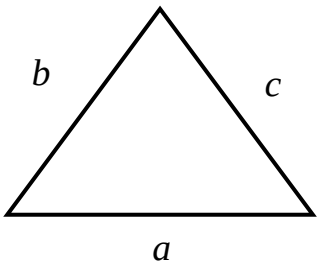


Рисунок 1.12 – Трикутник, для якого необхідно обчислити площу

Текст програми для обчислення площі трикутника за формулою Герона наведено на рис. 1.13.

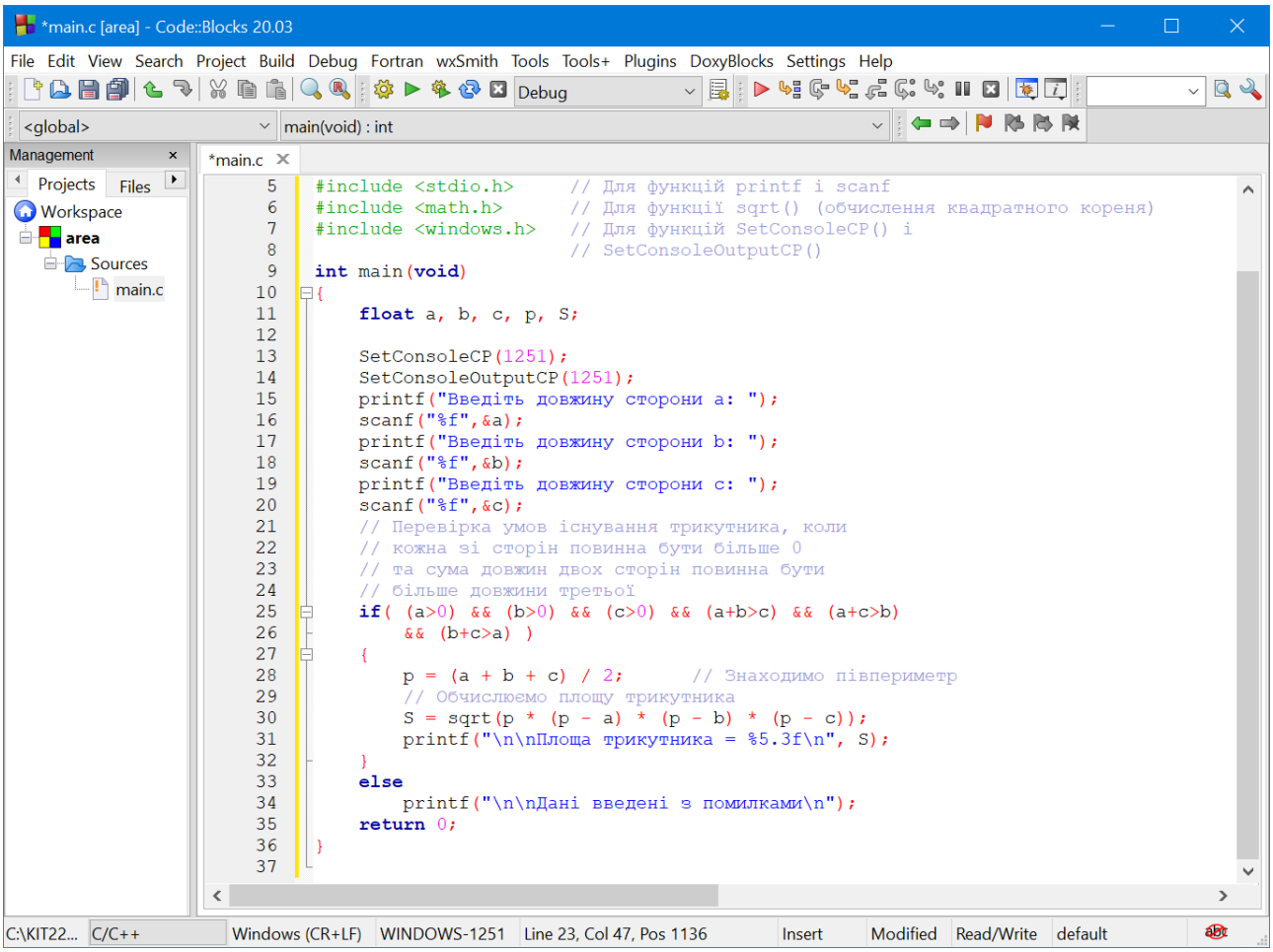


Рисунок 1.13 – Текст *C*-програми для обчислення площі трикутника

Формула Герона для обчислення площі трикутника має такий вигляд:

$$S = \sqrt{p(p-a)(p-b)(p-c)},$$

де S – площа трикутника; a, b, c – сторони трикутника; p – півпериметр трикутника, що обчислюється за формулою

$$p = \frac{a+b+c}{2}.$$

Для виконання програми необхідно натиснути іконку (рис. 1.14) на панелі «Compiler Toolbar» або скористатися комбінацію клавіш Ctrl+F10. Можна також вибрати відповідний рядок у пункті меню «Build->Run» (рис. 1.15). Результати роботи програми для значень сторін, які задовольняють умовам існування трикутника, наведені на рис. 1.16.

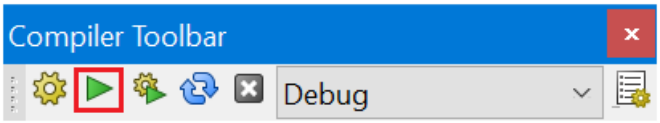


Рисунок 1.14 – Панель з кнопками для компонування та виконання програми

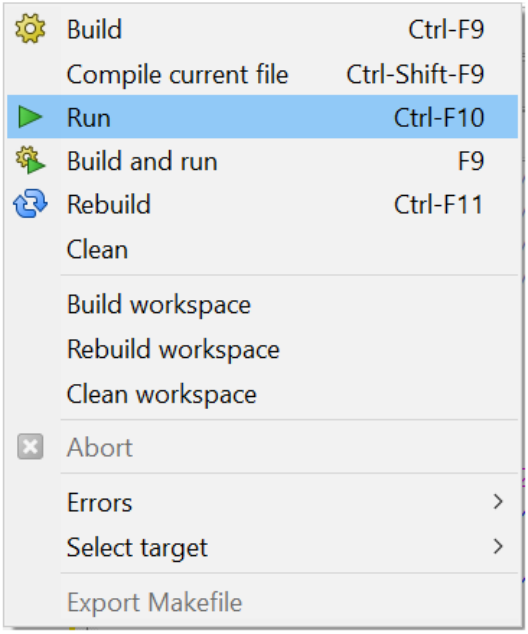


Рисунок 1.15 – Вибір пункту меню «Run» для запуску програми на виконання

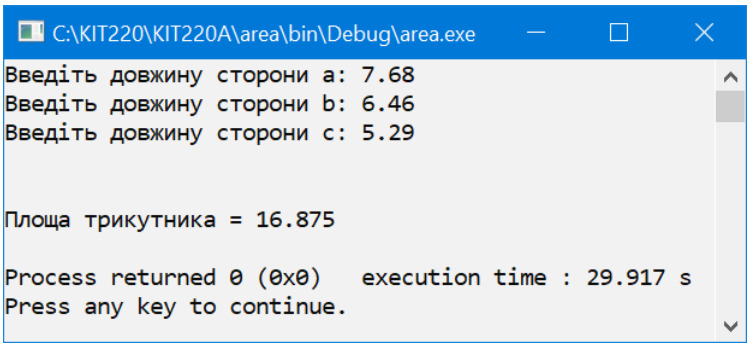


Рисунок 1.16 – Вікно консолі з результатами роботи програми

Хід виконання роботи:

1. Наберіть в редакторі інтегрованого середовища розробки *Code::Blocks* текст *C*-програми для обчислення площі трикутника за формулою Герона. Збережіть набраний текст за допомогою комбінації клавіш Ctrl+S або натисніть на іконку .

2. Запустіть програму на виконання (Ctrl+F10).

3. Якщо за результатами компіляції програми будуть виявлені помилки, зробіть копію екрану з зазначеними помилками та спробуйте їх за можливістю знайти та виправити.

4. Якщо програма не буде містити помилок, наведіть у звіті про виконання лабораторної роботи текст програми (шрифт – «Courier New», розмір – 12 пт, міжрядковий інтервал – «одинарний»).

5. Введіть необхідні дані для розрахунку площі трикутника з урахуванням умов існування трикутника. Копію екрана консолі з отриманим результатом додайте до звіту.

6. Введіть дані, що суперечать умовам існування трикутника, та наведіть відповідну копію екрана консолі.

7. Навмисно зробіть в тексті *C*-програми послідовно три помилки. В кожному випадку наведіть копію екрана з повідомленнями компілятора про наявність помилки.

8. Проекспериментуйте з текстом програми, закоментувавши спочатку рядок «`SetConsoleCP(1251);`» (поставивши попереду рядка //), а потім – «`SetConsoleOutputCP(1251);`». Поясніть у висновках по роботі, що буде відбуватися у кожному з цих випадків при запуску програми на виконання.

9. Напишіть невеличку *C*-програму, яка виводить на консоль довільний україномовний текст в чотири рядки. Потім пропустіть один рядок і на наступному вкажіть своє прізвище та ініціали, а також зазначте номер групи. Зробіть копію екрана з результатом роботи програми.

Примітка: Вивести довільний текст можна за допомогою оператора «`printf("Україномовний текст\n");`». Символ `'\n'` означає перехід на новий рядок. Для того щоб пропустити один рядок, треба в четвертому рядку для «`printf();`» поставити два таких символи.

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:
   – тексти двох програм (відповідно до вимог оформлення);
   – усі копії екранів, які необхідно додати до звіту (перелік зазначений в ході виконання роботи).

Всі копії екранів повинні бути оформлені як рисунки з відповідними підписами під ними (наприклад: Рисунок 1.1 – Копія екрану з результатами роботи програми). Рисунки слід оформляти на зразок тих, що наведені в даній лабораторній роботі. На кожен з рисунків повинно бути посилання в тексті звіту.

4. Розгорнуті висновки по роботі (не переписувати мету роботи).

Контрольні запитання:

1. Що собою являє Code::Blocks?
2. Як створити новий проєкт?
3. Який рядок програми відповідає за вивід україномовного тексту?
4. Які налаштування можна зробити для вікна консолі?
5. Назвіть умови існування трикутника для коректного обчислення його площі за формулою Герона.

ЛАБОРАТОРНА РОБОТА №2
ДОСЛІДЖЕННЯ ОСНОВНИХ ПРИНЦИПІВ АЛГОРИТМІЗАЦІЇ ОБЧИСЛЮВАЛЬНИХ ПРОЦЕСІВ

Мета роботи: розглянути основні етапи обчислення значення арифметичного виразу з урахуванням вхідних даних; ознайомитися з різними формами подання алгоритму; навчитися записувати алгоритм рішення задачі на псевдокоді; побудувати блок-схему розгалуженого алгоритму; провести трасування алгоритму для різних вхідних значень; навчитися записувати алгоритм у вигляді *C*-програми.

Теоретична частина

Розглянемо основні етапи обчислення значення арифметичного виразу $y = \frac{\sqrt{x^2-4}}{x-8}$ з урахуванням вхідних даних.

1. Для вирішення поставленого завдання необхідно визначити область припустимих значень даної функції.

Оскільки функція являє собою дріб, необхідно щоб, перш за все, знаменник не дорівнював нулю, тобто $(x-8) \neq 0$. Чисельник, у свою чергу, представлений у вигляді квадратного кореня. Якщо подивитися на графік функції $y = \sqrt{x}$ (рис. 2.1), можна побачити, що змінна x може приймати лише додатні значення. Відповідно підкореневий вираз не може бути менше нуля, тобто повинна виконуватися умова $x^2 - 4 \geq 0$ або $x^2 \geq 4$.

2. Визначивши необхідні умови, можна записати алгоритм знаходження значень функції на псевдокоді:

```
1) Ввести значення змінної x з клавіатури.
2) Якщо (x − 8) ≠ 0 та
3)   якщо x² ≥ 4, то
4)         обчислити y = √(x²−4) / (x−8),
5)         вивести значення y.
6)   інакше "Підкореневий вираз менше нуля. Немає рішення".
7) інакше "Знаменник дорівнює нулю. Немає рішення".
```

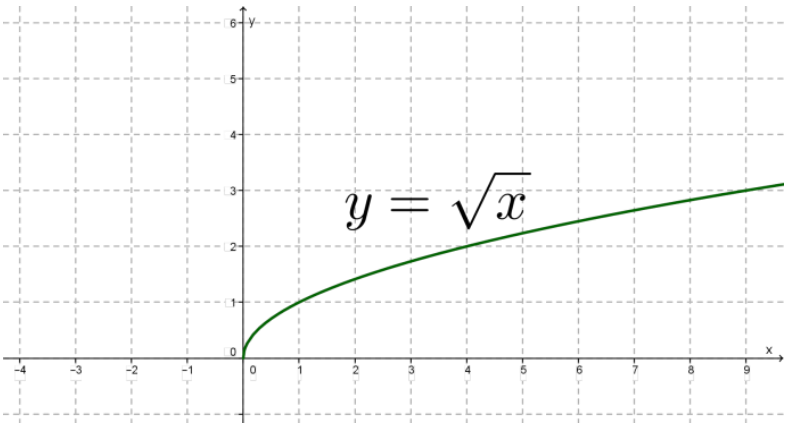


Рисунок 2.1 – Графік функції $y = \sqrt{x}$

3. Блок-схему алгоритму для даної задачі наведено на рис. 2.2 [3].

4. Проведемо трасування (покрокове виконання) цього розгалуженого алгоритму, припустивши, що користувач ввів з клавіатури число 3 (табл. 2.1).

Таблиця 2.1 – Трасування розгалуженого алгоритму ($x = 3$)

№ кроку	№ блоку	x	y	Примітки
1	2	3		Вводимо x з клавіатури
2	3			$x - 8 \neq 0$ – так
3	4			$x^2 \geq 4$ – так
4	5		–0.4472136	Обчислюємо $y = \frac{\sqrt{x^2-4}}{x-8}$
5	6			Виводимо значення y

Розглянемо тепер дії алгоритму, якщо користувач введе число 1 (табл. 2.2).

Таблиця 2.2 – Трасування розгалуженого алгоритму ($x = 1$)

№ кроку	№ блоку	x	y	Примітки
1	2	1		Вводимо x з клавіатури
2	3			$x - 8 \neq 0$ – так
3	4			$x^2 \geq 4$ – ні
4	7			Виводимо: «Підкореневий вираз менше нуля. Немає рішення»

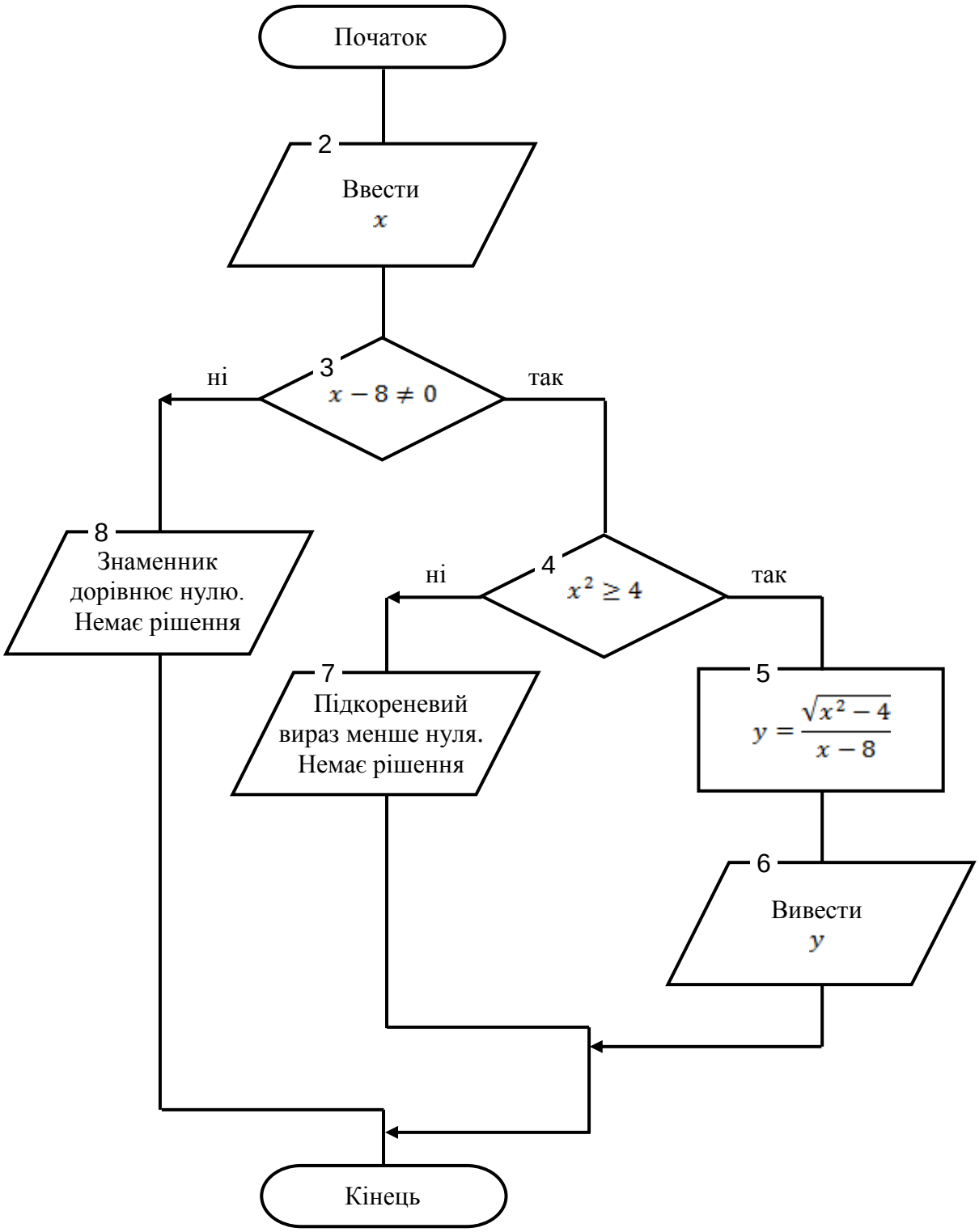


Рисунок 2.2 – Блок-схема розгалуженого алгоритму

Розглянемо тепер дії алгоритму, якщо користувач введе число 8 (табл. 2.3).

Таблиця 2.3 – Трасування розгалуженого алгоритму ($x = 8$)

№ кроку	№ блоку	x	y	Примітки
1	2	8		Вводимо x з клавіатури
2	3			$x - 8 \neq 0$ – ні
3	8			Виводимо: «Знаменник дорівнює нулю. Немає рішення»

5. С-програма, яка реалізує даний алгоритм має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <math.h>

int main(void)
{
    float x, y;

    SetConsoleOutputCP(1251);

    printf("Введіть x: ");
    scanf("%f", &x);

    if(x - 8 != 0)
    {
        if(x * x >= 4)
        {
            y = sqrtf(x * x - 4) / (x - 8);
            printf("\nЗначення функції y = %9.7f\n", y);
        }
        else
        {
            printf("\nПідкореневий вираз менше нуля. "
                   "Немає рішення\n");
        }
    }
    else
    {
        printf("\nЗнаменник дорівнює нулю. Немає рішення\n");
    }
    return 0;
}
```

Результат виконання даної програми для $x = 3$ наведено на рис. 2.3, для $x = 1$ – на рис. 2.4, для $x = 8$ – на рис. 2.5.

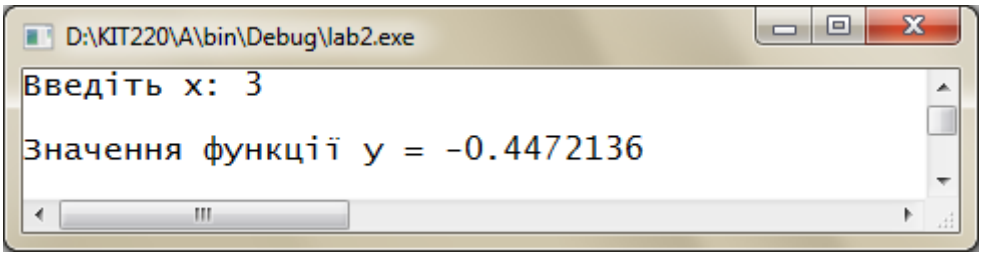


Рисунок 2.3 – Результат виконання програми для $x = 3$

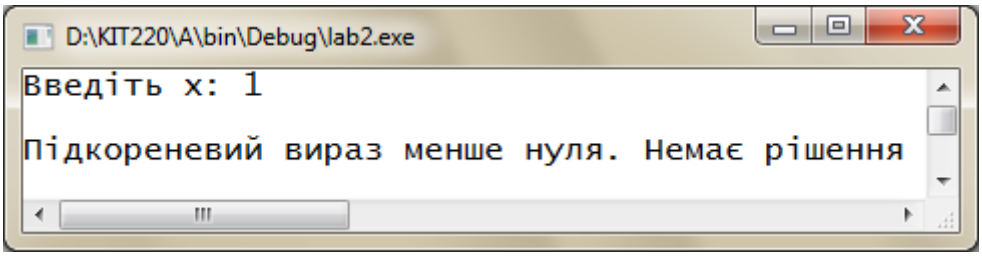


Рисунок 2.4 – Результат виконання програми для $x = 1$

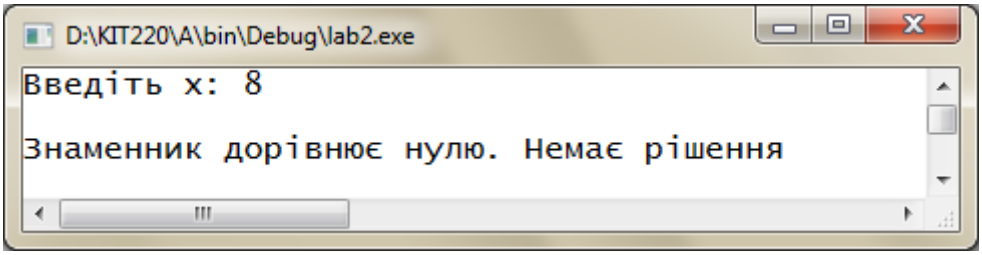


Рисунок 2.5 – Результат виконання програми для $x = 8$

Цю задачу можна оптимізувати, об'єднавши дві умови в одну. Оскільки ці умови повинні виконуватися одночасно, їх необхідно об'єднати за допомогою логічного «І» (на мові *C* – **&&**).

Тоді псевдокод буде виглядати наступним чином:

1) Ввести число x з клавіатури.
2) Якщо $x \neq 8$ і $x^2 \geq 4$, то
3) обчислити $y = \frac{\sqrt{x^2-4}}{x-8}$,
4) вивести значення y.
5) інакше "Немає рішення".

Блок-схема модифікованого алгоритму наведена на рис. 2.6.

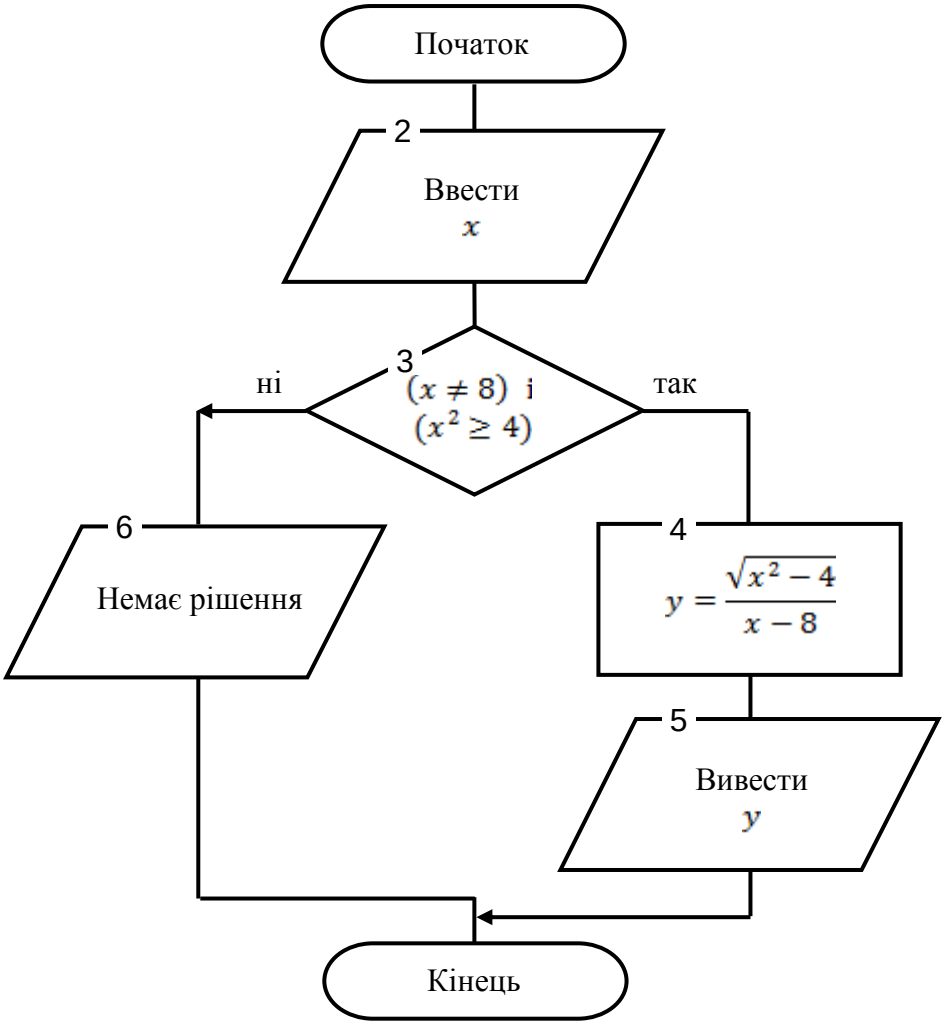


Рисунок 2.6 – Блок-схема модифікованого алгоритму

С-програма модифікованого алгоритму має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <math.h>
int main(void)
{
    float x, y;
    SetConsoleOutputCP(1251);
    printf("Введіть x: ");
    scanf("%f", &x);
    if( (x != 8) && (x * x >= 4) )
    {
        y = sqrtf(x * x - 4) / (x - 8);
        printf("\nЗначення функції y = %9.7f\n", y);
    }
    else printf("\nНемає рішення\n");
    return 0;
}
```

Хід виконання роботи:

1. Відповідно до виразу для вашого варіанта (табл. 2.4), надайте словесний опис алгоритму рішення задачі.

Таблиця 2.4 – Арифметичні вирази

№ варіанта	Вираз	№ варіанта	Вираз
1	$y = \frac{\sin(1/x)}{\sqrt{x+1}}$	11	$y = \frac{\log_2 x}{x-3}$
2	$y = \frac{3x}{-x^2+7x+8}$	12	$y = \frac{2\cos x}{-x^2+2x+8}$
3	$y = \frac{\ln x}{x-1}$	13	$y = \frac{\sqrt{\sin x}}{x-2}$
4	$y = \frac{\sin x}{x^2+8x+15}$	14	$y = \frac{2\cos x}{x^2+5x-6}$
5	$y = \frac{\sqrt{\cos x}}{x}$	15	$y = \frac{x-3}{2x^2-3x+1}$
6	$y = \frac{x+1}{15-2x-x^2}$	16	$y = \frac{2x}{x^2-2x-3}$
7	$y = \frac{1}{\ln(x+3)}$	17	$y = \sqrt{x}\ln(9-x^2)$
8	$y = \frac{1}{\sqrt{x^2+4x+3}}$	18	$y = \frac{\lg x}{x-2}$
9	$y = -\frac{2}{4-x^2}$	19	$y = \frac{1}{\sqrt{3x+2}}$
10	$y = \frac{2x+1}{x^2+3x+2}$	20	$y = \frac{2\sin x}{x^2-5x+6}$

Графіки деяких математичних функцій для визначення області припустимих значень наведені на рис. 2.7.

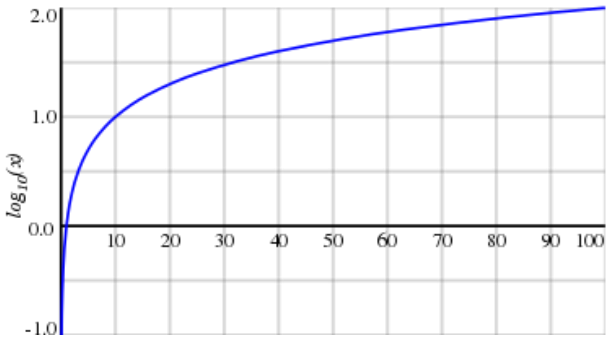
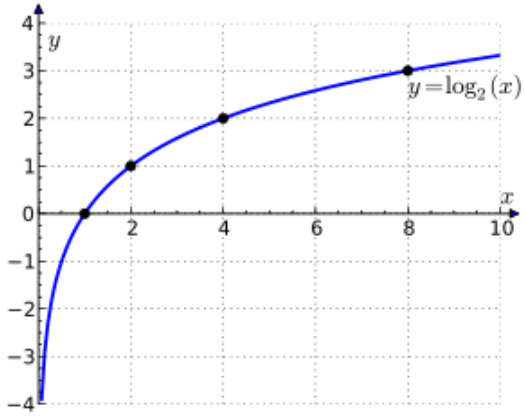
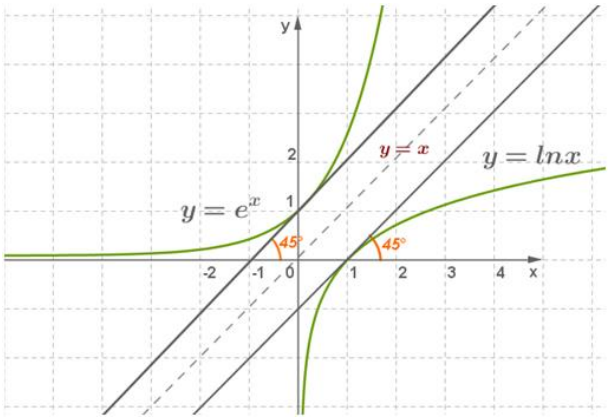
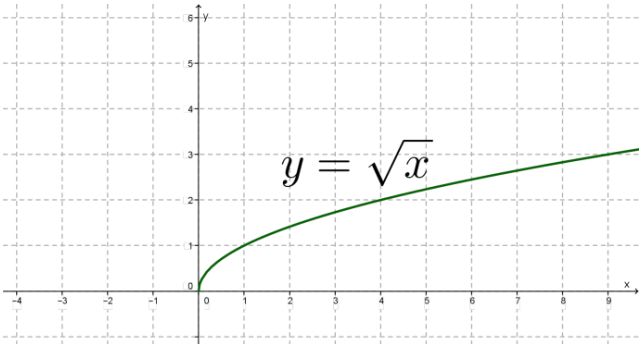
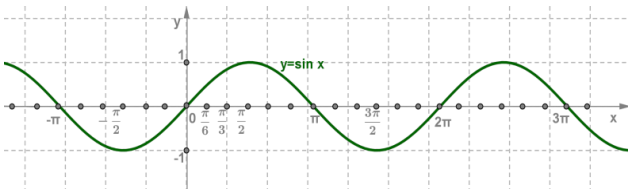
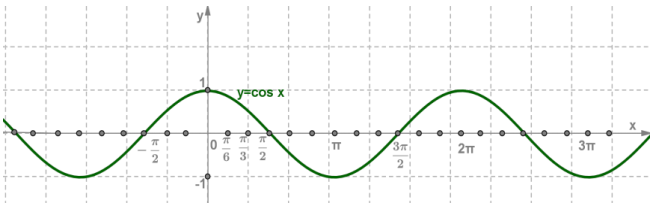
 Графік функції $y = \lg x$	 Графік функції $y = \log_2 x$
Область визначення: $0 < x < +\infty$. Множина значень: $-\infty < y < +\infty$.	Область визначення: $0 < x < +\infty$. Множина значень: $-\infty < y < +\infty$.
 Графік функції $y = \ln x$	 Графік функції $y = \sqrt{x}$
Область визначення: $0 < x < +\infty$. Множина значень: $-\infty < y < +\infty$.	Область визначення: $0 \leq x < +\infty$. Множина значень: $0 \leq y < +\infty$.
 Графік функції $y = \sin x$	 Графік функції $y = \cos x$
Область визначення: $-\infty < x < +\infty$. Множина значень: $-1 \leq y \leq +1$.	Область визначення: $-\infty < x < +\infty$. Множина значень: $-1 \leq y \leq +1$.

Рисунок 2.7 – Графіки деяких математичних функцій

2. Наведіть алгоритм на псевдокоді.

3. Побудуйте блок-схему алгоритму.

4. Зробіть трасування алгоритму для основних гілок у вигляді таблиць.

5. Напишіть *C*-програму, яка відповідає блок-схемі алгоритму.

6. Виконайте програму для кожної гілки алгоритму.

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.

2. Мету роботи.

3. Результати виконання роботи:

   – вираз, який необхідно обчислити;

   – словесний опис алгоритму;

   – запис алгоритму на псевдокоді;

   – блок–схему алгоритму;

   – трасування всіх гілок алгоритму у вигляді таблиць;

   – *C*–програму, яка реалізує заданий алгоритм;

   – копії екранів консолі з результатами виконання програми для кожної гілки алгоритму.

4. Висновки по роботі.

Контрольні запитання:

1. Дайте визначення поняттю «алгоритм».

2. Перерахуйте основні властивості алгоритмів і розкрийте їх сутність.

3. Як поділяються алгоритми за типом реалізованого обчислювального процесу?

4. Які способи опису алгоритмів вам відомі?

5. Що розуміється під графічним способом опису алгоритмів? У чому полягає перевага даного способу перед словесним описом алгоритму?

6. Що таке трасування алгоритму? Для чого воно призначене?

ЛАБОРАТОРНА РОБОТА №3
ДОСЛІДЖЕННЯ БАЗОВИХ ТИПІВ ДАНИХ МОВИ *C*

Мета роботи: ознайомитися з базовими типами даних, які використовуються у мові *C*; навчитися визначати об'єм пам'яті для кожного з базових типів; відпрацювати вивід на консоль значень різних типів даних за допомогою функції `printf()`.

Теоретична частина

3.1. Ключові слова для типів даних

У мові *C* передбачено декілька базових типів даних [4]. Розглянемо ключові слова для базових типів даних, які розпізнаються мовою *C*.

У стандарті K&R *C* існувало сім ключових слів (табл. 3.1), які мали відношення до типів. У стандарті C89 до цього списку були додані ще два ключових слова. У стандарті C99 список поповнився ще трьома ключовими словами.

Таблиця 3.1 – Ключові слова для типів даних мови *C*

Ключові слова в стандарті K&R *C*	Ключові слова, які додані стандартом C89	Ключові слова, які додані стандартом C99
`int`	**`signed`**	**`_Bool`**
`long`	**`void`**	**`_Complex`**
`short`		**`_Imaginary`**
`unsigned`		
`char`		
`float`		
`double`		

Ключовим словом **`int`** позначається основний клас цілих чисел, що застосовуються в *C*. Наступні три ключових слова (**`long`**, **`short`** і **`unsigned`**) і додане стандартом C89 ключове слово **`signed`** використовуються для вказівки варіацій цього базового типу, наприклад, **`unsigned short int`** і **`long long int`**.

За допомогою ключового слова **char** визначаються символьні дані, до яких відносяться літери алфавіту й інші символи, такі як '#', '$', '%' і '*'. Тип даних **char** можна також застосовувати для представлення невеликих цілих чисел, яким для двійкового представлення достатньо використовувати 8 бітів.

Типи **float**, **double** і **long double** призначені для представлення чисел з рухомою комою. Тип даних **_Bool** використовується для булевих значень (true і false), а типи даних **_Complex** і **_Imaginary** представляють відповідно комплексні та уявні числа.

Типи даних, що створюються за допомогою зазначених ключових слів, можуть бути розділені на два сімейства на основі того, як вони зберігаються в пам'яті комп'ютера: цілочислові типи та типи з рухомою комою.

3.2. Біти, байти та слова

Для опису елементів комп'ютерних даних або елементів комп'ютерної пам'яті можуть застосовуватися терміни «біт», «байт» і «слово». Другому терміну приділяється основна увага.

Біт – це мінімальна одиниця пам'яті. Він є базовим будівельним блоком для пам'яті комп'ютера і може зберігати одне з двох значень: 0 або 1.

Байт – це найбільш часто використовувана одиниця пам'яті комп'ютера. Практично на всіх машинах байт складається з 8 бітів, і це є стандартним визначенням байта, принаймні, коли мова йде про вимірювання об'єму пам'яті. Оскільки біт може приймати значення 0 або 1, байт забезпечує 256 (тобто 2^8) можливих комбінацій нулів і одиниць. Ці комбінації можуть використовуватися, наприклад, для представлення цілих чисел від 0 до 255 або набору символів. Числа можна записувати за допомогою двійкового коду, в якому для представлення чисел застосовуються тільки нулі й одиниці.

Слово – це природна одиниця пам'яті для комп'ютера конкретного типу. У 8-розрядних комп'ютерах, таких як перші машини Apple, слово складалося з 8 бітів. З тих пір персональні комп'ютери перейшли на 16-бітні, 32-бітні, а в

даний час і 64-бітні слова. Великі розміри слова дозволяють швидше передавати дані та надають доступ до більшого об’єму пам’яті.

3.3. Цілі числа

Ціле число – це число без дробової частини. У мові *C* ціле число ніколи не записується з десятковою крапкою, наприклад, 2, –23, 2456. Числа на зразок 3.14, 0.22 і 2.000 не є цілими. Цілі числа зберігаються в двійковій формі. Наприклад, ціле число 7 записується в двійковій формі як 111_2. Отже, щоб зберегти це число у 8-бітному байті, потрібно просто встановити перші 5 бітів в 0, а останні три біти – в 1 (рис. 3.1).

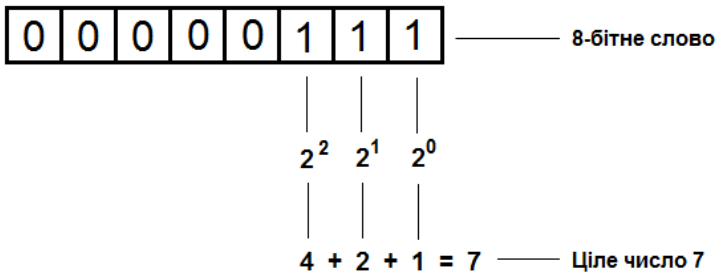


Рисунок 3.1 – Зберігання цілого числа 7 у двійковому коді

3.4. Числа з рухомою комою

Число з рухомою комою більш-менш відповідає тому, що математики називають дійсним числом. До дійсних чисел відносяться числа, які знаходяться в проміжках між цілими числами. Прикладами чисел з рухомою комою можуть служити 2.75, 3.16E7, 7.00 і 2e–8. Зверніть увагу, що додавання десяткового дробу перетворює ціле число на число з рухомою комою. Таким чином, 7 має цілочисловий тип, але 7.00 – тип з рухомою комою. Очевидно, що існує більше однієї форми запису числа з рухомою комою. Запис 3.16E7 означає, що число 3.16 необхідно помножити на 10^7. Число 7 називається порядком числа 10.

Ключовим моментом тут є те, що схема, яка використовується для зберігання числа з рухомою комою, відрізняється від схеми, яка застосовується для зберігання цілого числа. Число з рухомою комою розділяється на дробову час-

тину і порядок, які зберігаються окремо. Таким чином, 7.00 буде зберігатися в пам'яті не в тому вигляді, в якому зберігається ціле число 7, хоча обидва вони мають одне й те ж саме значення.

Десятковим аналогом запису 7.00 може бути 0.7E1, де 0.7 – дробова частина числа, а 1 – порядок. На рис. 3.2 наведено ще один приклад зберігання числа з рухомою комою.

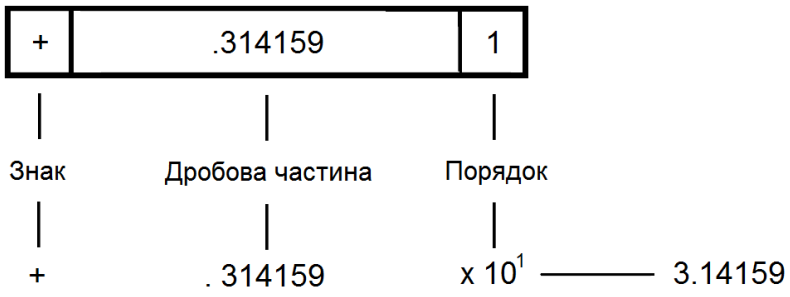


Рисунок 3.2 – Зберігання числа π у форматі з рухомою комою (десяткова версія)

Зрозуміло, що для зберігання комп'ютер використовує двійкові числа і степені 2, а не степені 10.

3.5. Базові типи даних мови *C*

Давайте поглянемо на особливості базових типів даних, що використовуються в мові *C*.

3.5.1. Тип `int`

Мова *C* пропонує декілька цілочислових типів, надаючи програмісту можливість зіставлення типу з конкретним випадком використання. Зокрема, цілочислові типи *C* варіюються в діапазонах допустимих значень і в можливості застосування від'ємних чисел. Тип **`int`** є базовим вибором, але якщо будуть потрібні інші варіанти, що задовольняють вимогам певної задачі або комп'ютера, то вони також доступні.

Тип **int** представляє собою ціле число зі знаком. Це означає, що воно повинно бути цілим і може мати додатну, від'ємну або нульову величину. Діапазон можливих значень залежить від комп'ютерної системи. Зазвичай для зберігання даних типу **int** використовується одне машинне слово. Тому в комп'ютерах, сумісних зі старими моделями IBM PC з 16-бітними словами, для зберігання даних типу **int** виділялося 16 бітів. Це дозволяло мати діапазон значень від –32768 до 32767. Сучасні персональні комп'ютери зазвичай оперують 32-бітними цілими числами і дані типу **int** відповідають такому розміру. В наш час індустрія персональних комп'ютерів зорієнтувалася на випуск 64-розрядних процесорів, які можуть вільно маніпулювати ще більшими цілими числами. У стандарті ISO *C* вказано, що мінімальним діапазоном для типу **int** повинен бути діапазон від –32768 до 32767. Зазвичай системи представляють цілі числа зі знаком за рахунок використання значення певного біта.

3.5.1.1. Оголошення змінної типу `int`

Для оголошення цілочислових змінних застосовується ключове слово **int**. Спочатку вказується ключове слово **int**, потім обране ім'я для змінної і, нарешті, крапка з комою. Оголошення декількох змінних можна робити або окремо, або помістивши після ключового слова **int** список імен, відокремлюючи їх комами. Нижче наведені приклади допустимих оголошень:

```
int ems;
int hogs, cows, goats;
```

Для кожної змінної можна було б передбачити окреме оголошення або ж оголосити всі чотири змінні в одному операторі. Результат був би таким самим: зв'язування імен з виділеними областями пам'яті для чотирьох змінних типу **int.** Ці оголошення створюють змінні, але не надають їм значення. Яким чином змінні отримують значення? Розглянемо способи, за допомогою яких змінні можуть отримувати значення в програмі.

Перший спосіб – це оператор присвоювання:

```
cows = 112;
```

Другий спосіб передбачає отримання змінною значення з функції, наприклад, з функції `scanf()`.

Третій спосіб – ініціалізація змінних.

3.5.1.2. Ініціалізація змінних типу `int`

Ініціалізація змінної означає присвоювання їй початкового значення. У мові *C* це можна робити під час оголошення змінної. Досить після імені змінної помістити операцію присвоювання `(=)` і вказати значення, яке змінна повинна отримати. Наприклад:

```
int hogs = 21;
int cows = 32, goats = 14;
int dogs, cats = 94;          // допустима, але невдала форма
```

В останньому рядку ініціалізується тільки змінна `cats`. Однак через неуважність може здатися, що змінна `dogs` також ініціалізується значенням 94, тому краще уникати використання в одному операторі оголошення ініціалізованих і неініціалізованих змінних.

Оголошення виділяють і позначають для змінних області зберігання, а також надають їм початкові значення.

3.5.1.3. Вивід значень типу `int`

Для виводу значень типу **`int`** можна застосовувати функцію `printf()`. Конструкція %d служить для зазначення в рядку місця, де буде виводитися ціле число. Конструкція %d називається специфікатором формату, оскільки вона визначає формат, який використовує функція `printf()` для відображення конкретного значення. Кожному специфікатору %d в рядку формату має відповідати значення **`int`** у списку виведених елементів. Таким значенням може бути змінна **`int`**, константа **`int`** або будь-який інший вираз **`int`.** Програміст повинен стежити за тим, щоб кількість специфікаторів формату відповідала числу значень, тому що компілятор не виявляє помилки подібного роду.

Далі представлена програма, яка ініціалізує змінну, а потім виводить значення цієї змінної, значення константи та значення простого виразу. До того ж вона демонструє, що відбувається у разі неуважності.

```
#include <stdio.h>
#include <windows.h>

int main(void)
{
    int ten = 10;
    int two = 2;

    SetConsoleOutputCP(1251);

    printf("%d мінус %d дорівнює %d\n", ten, 2, ten - two);
    return 0;
}
```

Результат виконання програми наведено на рис. 3.3.

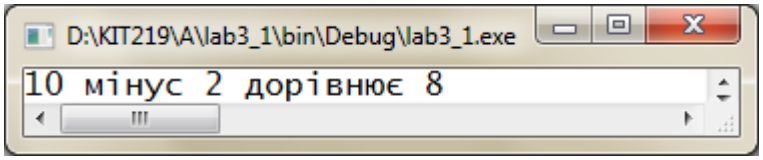


Рисунок 3.3 – Результат виконання програми для виводу на консоль даних, які мають цілочисловий тип

У рядку виводу перший специфікатор %d являє собою змінну **`ten`** типу **`int`**, другий – константу 2 типу **`int`** і третій – значення виразу `ten - two` типу **`int`**.

Для запобігання можливих помилок, найкраще завжди перевіряти в функції `printf()` відповідність кількості специфікаторів формату кількості значень, які підлягають відображенню на екрані.

3.5.1.4. Вісімкові та шістнадцяткові числа

Зазвичай в мові *C* передбачається, що цілочислові константи є десятковими числами (з основою 10). Однак у багатьох програмістів користуються популярністю вісімкові (з основою 8) та шістнадцяткові (з основою 16) числа. Оскільки 8 і 16 являють собою степені числа 2, а 10 – ні, то вісімкова та шістнадцяткова системи числення є більш зручними для представлення чисел, що пов’язані з комп’ютерами.

Наприклад, число 65536_{10}, яке часто зустрічається в 16-розрядних машинах, в шістнадцятковій формі записується як 10000_{16}. Крім того, кожна цифра шістнадцяткового числа відповідає в точності 4 бітам. Наприклад, шістнадцяткова цифра 3_{16} – це 0011_2, а шістнадцяткова цифра 5_{16} – це 0101_2. Таким чином, шістнадцятковому значенню 35_{16} відповідає бітова комбінація $0011\ 0101_2$, а шістнадцятковому значенню 53_{16} – $0101\ 0011_2$. Така відповідність дозволяє полегшити перехід від шістнадцяткового представлення числа до двійкового представлення (з основою 2) і навпаки. Але яким чином комп'ютер може визначити, в якій формі записано число 10000 – в десятковій, шістнадцятковій або вісімковій? В мові *C* система числення задається за допомогою спеціального префікса. Префікс 0x або 0X означає, що вказується шістнадцяткове число, тому 16 в шістнадцятковому вигляді записується як 0x10 або 0X10. Аналогічно, префікс 0 (нуль) означає, що задається вісімкове число. Наприклад, в *C* десяткове число 16 у вісімковому вигляді записується як 020.

Можливість застосування різних систем числення є лише зручністю для програмістів. Це не впливає на спосіб зберігання числа в пам'яті. Іншими словами, можна написати 16, 020 або 0x10, і це число в кожному випадку буде зберігатися в пам'яті однаковим чином – у двійковому коді, який використовується всередині комп'ютера.

3.5.1.5. Відображення вісімкових і шістнадцяткових чисел

Мова *C* дозволяє не тільки записувати число в кожній з трьох систем числення, а й відображати його в кожній з перерахованих систем числення. Щоб вивести на екран ціле число у вісімковому, а не десятковому вигляді, замість %d застосовуйте специфікатор %o. Для відображення цілого числа в шістнадцятковому вигляді служить специфікатор %x. Якщо необхідно вивести на екран префікси мови *C*, скористайтеся специфікаторами %#o, %#x і %#X, які дозволяють відображати префікси 0, 0x і 0X.

Розглянемо невеличкий приклад програми, яка дозволяє виводити на консоль вісімкові та шістнадцяткові числа. Її текст має такий вигляд:

```
#include <stdio.h>
#include <windows.h>

int main(void)
{
    int x = 100;

    SetConsoleOutputCP(1251);

    printf("десяткове      = %d\n"
           "вісімкове      = %o\n"
           "шістнадцяткове = %x\n\n\n", x, x, x);
    printf("десяткове      = %d\n"
           "вісімкове      = %#o\n"
           "шістнадцяткове = %#x\n", x, x, x);
    return 0;
}
```

Результат виконання програми наведено на рис. 3.4.

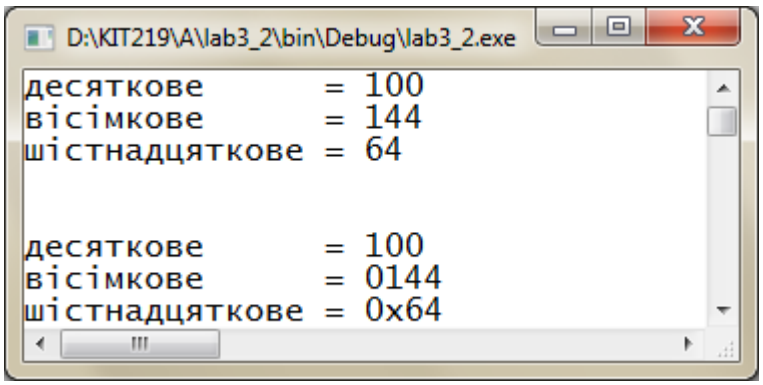


Рисунок 3.4 – Результат виконання програми для виводу на консоль цілочислових даних у різних системах числення

3.5.1.6. Інші цілочислові типи

У мові *C* застосовуються три ключових слова, що модифікують базовий цілочисловий тип: **`short`**, **`long`** і **`unsigned`**. При цьому, слід взяти до уваги такі аспекти:

– тип **`short int`** або **`short`** може використовувати менший об’єм пам’яті, ніж **`int`**, і тим самим зекономити пам’ять у разі, коли потрібні тільки невеликі числа. Подібно до **`int`**, тип **`short`** є типом зі знаком;

– тип **long int** або **long** може займати більший об'єм пам'яті, ніж **int**, дозволяючи представляти великі цілочислові значення. Подібно до **int**, тип **long** є типом зі знаком;

– тип **long long int** або **long long** (введений стандартом С99) може займати більше пам'яті, ніж **long**. Для цього типу використовуються мінімум 64 біта. Подібно до **int**, тип **long long** є типом зі знаком;

– тип **unsigned int** або **unsigned** застосовується для змінних, які приймають тільки додатні значення. Цей тип зсовує діапазон збережених чисел. Наприклад, 16-бітний тип **unsigned int** має діапазон значень від 0 до 65535 замість діапазону від –32768 до 32767. Біт, який використовувався для представлення знаку, тепер стає ще однією двійковою цифрою, роблячи можливим представлення більшого числа;

– типи **unsigned long int** або **unsigned long** і **unsigned short int** або **unsigned short** розпізнаються як допустимі стандартом С89. У стандарті С99 до них додано тип **unsigned long long int** або **unsigned long long**;

– ключове слово **signed** може застосовуватися з будь-якими типами зі знаком, щоб явно вказати свій намір. Наприклад, **short**, **short int**, **signed short** і **signed short int** є іменами одного й того ж самого типу.

3.5.1.7. Оголошення змінних інших цілочислових типів

Змінні інших цілочислових типів оголошуються в такій самій манері, що й змінні типу **int**. Нижче наведено список прикладів таких оголошень. Не всі старі компілятори *C* розпізнають останні три оголошення, а фінальне оголошення відповідає стандарту С99.

```
long int            estine;
long                Johns;
short int           erns;
short               ribs;
unsigned int        s_count;
unsigned            players;
unsigned long       headcount;
unsigned short      yesvotes;
long long           ago;
```

3.5.1.8. Цілочислове переповнення

Що станеться, якщо ціле число виявиться більшим, ніж допускає обраний для нього тип? Дамо цілочисловій змінній максимально можливе ціле значення та додамо до нього ще якесь ціле число та подивимося, до чого це призведе. Виконаємо цю дію над типами зі знаком і без знаку. У виклику функції `printf()` для відображення значень типу **`unsigned int`** застосовується специфікатор %u.

```
#include <stdio.h>
#include <windows.h>

int main(void)
{
    int i = 2147483647;
    unsigned int j = 4294967295;

    SetConsoleOutputCP(1251);

    printf("%11d %11d %11d\n", i, i+1, i+2);
    printf("%11u %11u %11u\n", j, j+1, j+2);
    return 0;
}
```

Результат виконання програми наведено на рис. 3.5.

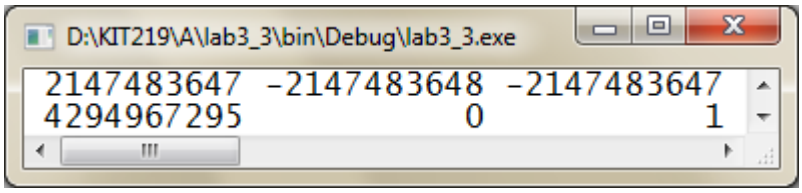


Рисунок 3.5 – Результат виконання програми для виводу на консоль цілочислових даних зі знаком і без знаку

Цілочислова змінна без знаку **`j`** діє як лічильник пробігу автомобіля. Коли досягається максимальне значення, воно скидається, і підрахунок починається з початку. Цілочислова змінна **`i`** поводиться аналогічним чином. Головна відмінність між ними полягає в тому, що значення змінної **`j`** типу **`unsigned int`**, подібно лічильнику пробігу, починаються з 0, в той самий час як значення змінної **`i`** типу **`int`** – з –2147483648. Зверніть увагу, що про перевищення максимально-

го значення (переповнення) змінної **i** нічого не повідомляється. Щоб відстежувати це, доведеться самостійно передбачити відповідний код.

Описана вище поведінка програми регламентується правилами мови *C* для беззнакових типів. У стандарті не визначено, як повинні поводитися типи зі знаком. Показана тут поведінка є типовою, але цілком можна зіткнутися з іншою поведінкою.

3.5.1.9. Вивід значень типів **short**, **long**, **long long** і **unsigned**

Для виводу чисел типу **unsigned int** застосовуйте специфікатор %u. Щоб вивести значення типу **long**, використовуйте специфікатор формату %ld. Якщо типи **int** і **long** у вашій системі мають однаковий розмір, цілком достатньо використовувати специфікатор %d, проте програма некоректно працюватиме при перенесенні до системи, де ці два типи мають різні розміри, тому для **long** краще застосовувати специфікатор %ld. Разом з префіксами `'x'` і `'o'` можна також вказувати префікс `'l'`. Таким чином, можна використовувати специфікатор %lx для виводу цілого числа типу **long** в шістнадцятковому форматі та специфікатор %lo – для його виводу у вісімковому форматі. Хоча мова *C* дозволяє застосовувати в якості суфіксів констант і прописні, і малі літери, в цих специфікаторах формату використовуються тільки малі літери.

У мові *C* доступні додаткові формати для `printf()`. Насамперед, можна застосовувати префікс `'h'` для значень типу **short**. Отже, специфікатор %hd відображає ціле число типу **short** в десятковій формі, а специфікатор %ho відображає це ж саме число у вісімковій формі. Префікси `'h'` і `'l'` можна використовувати разом з префіксом `'u'` для типів без знаку. Наприклад, для виводу значень типів **unsigned long** можна було б вказати %lu.

Розглянемо приклад програми, що призначена для систем, які підтримують типи **long long**. Для версій зі знаком і без знака застосовуються специфікатори %lld і %llu. Вона буде мати такий вигляд:

```
#include <stdio.h>
#include <windows.h>
```

```
int main(void)
{
    // система з 32-бітним типом int і 16-бітним типом short
    unsigned int    un = 3000000000;
    short           end = 200;
    long            big = 65537;
    long long       verybig = 12345678908642;

    SetConsoleOutputCP(1251);

    printf("un = %u, але не %d\n", un, un);
    printf("end = %hd і %d\n", end, end);
    printf("big = %ld, але не %hd\n", big, big);
    printf("verybig = %lld, але не %ld\n", verybig, verybig);
    return 0;
}
```

Результат виконання програми наведено на рис. 3.6.

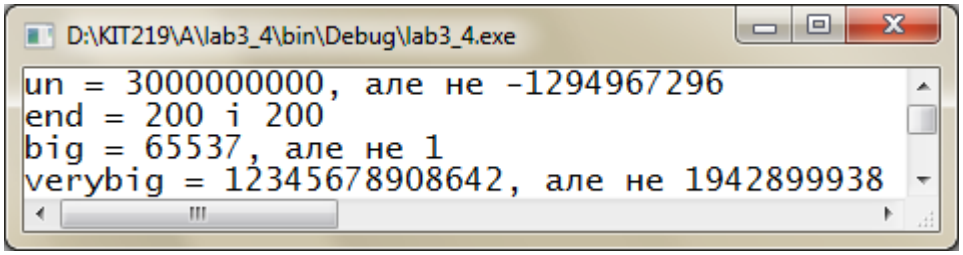


Рисунок 3.6 – Результат виконання програми для виводу на консоль цілочислових даних з різними специфікаторами формату

Цей приклад демонструє, що використання неправильних специфікаторів може призвести до несподіваних результатів. Перш за все, зверніть увагу, що застосування специфікатору %d для беззнакової змінної **un** видає від'ємне число. Причина цього в тому, що значення 3000000000 без знаку і значення –129496296 зі знаком мають одне й те ж саме внутрішнє представлення в пам'яті системи. Таким чином, якщо вказати функції `printf()`, що її параметром є число без знаку, вона виведе одне значення, а якщо вказати, що значенням є число зі знаком, – то інше значення. Подібна поведінка відбувається для значень, які перевищують максимально допустиме значення для типу зі знаком. Невеликі додатні значення, такі як 96, зберігаються та відображаються однаково як для типів зі знаком, так і для типів без знаку.

Далі відзначимо, що змінна **end** типу **short** відображається однаково незалежно від того, до якого типу вказується для функції `printf()` приналежність **end**: до типу **short** (специфікатор %hd) або до типу **int** (специфікатор %d). Це пояснюється тим, що при передачі аргументу функції *C* значення типу **short** автоматично розширюється до типу **int**. Тут можуть виникнути два питання: чому робиться вказане перетворення, і для чого використовується модифікатор `'h'`? Відповідь на перше питання є дуже простою: для типу **int** вибирався такий розмір, щоб забезпечити найбільш ефективну його обробку комп'ютером. Отже, на комп'ютері, в якому типи **short** і **int** мають різні розміри, передача значення як **int** може здійснюватися швидше.

Відповідь на друге питання виглядає так: модифікатор `'h'` можна застосовувати, щоб продемонструвати, який вид прийме ціле значення, будучи усіченим до типу **short**. Ілюстрацією цього твердження може служити третій рядок виводу. Число 65537, записане в двійковому форматі як 32-бітне число, має вигляд $00000000000000010000000000000001_2$. За допомогою специфікатора %hd функція `printf()` повинна переглядати тільки останні 16 бітів числа, тому вона відображає в результаті 1. Аналогічно, фінальний рядок виводу показує повне значення **verybig**, після чого це значення зберігається в останніх 32 бітах, на що вказує специфікатор %ld.

Таким чином програміст відповідає не тільки за те, щоб кількість специфікаторів відповідала кількості відображуваних значень, а й за правильний вибір специфікаторів, які відповідають типу відтворюваних значень.

3.5.2. Використання символів

Тип даних **char** застосовується для зберігання символів, таких як букви і розділові знаки, однак формально він також є цілочисловим. Причина в тому, що тип **char** в дійсності зберігає цілі числа, а не символи. Для підтримки символів комп'ютер використовує числовий код, в якому визначені цілі числа, які представляють певні символи. У США найбільш часто застосовується код

ASCII. Наприклад, ціле значення 65 в ньому представляє велику літеру `'A'`. Таким чином, щоб зберегти букву `'A'`, фактично потрібно записати ціле число 65.

Стандартний код ASCII складається з послідовності чисел від 0 до 127. Цей діапазон є досить малим, щоб для значення вистачило 7 бітів. Тип **`char`** зазвичай визначається як 8-бітна одиниця пам'яті, тому її більш ніж достатньо, щоб вмістити стандартний код ASCII. У багатьох системах, таких як IBM PC і Apple Macintosh, використовуються розширені коди ASCII (різні для цих двох систем), які як і раніше не виходять за межі 8 бітів. У загальному випадку мова *C* гарантує, що тип **`char`** є достатньо великим, щоб представляти базовий набір символів у системах, для яких реалізовані компілятори *C*.

Багато наборів символів містять більш ніж 127 або навіть 255 значень. Наприклад, існує набір символів Japanese kanji для японських ієрогліфів. В рамках комерційної ініціативи Unicode була створена система для представлення широкої різноманітності наборів символів, що застосовуються у різних частинах світу, яка в даний час містить понад 110000 символів. Організація ISO і комісія IEC (International Electrotechnical Commission – Міжнародна електротехнічна комісія) разом розробили для наборів символів стандарт, який отримав назву ISO/IEC 10646. На щастя, стандарт Unicode зберіг сумісність з більш широким стандартом ISO/IEC 10646.

Мова *C* визначає байт як кілька бітів, що використовуються типом **`char`**, тому може бути система з 16- або 32-бітним байтом і типом **`char`**.

3.5.2.1. Оголошення змінних типу `char`

Як і можна було очікувати, змінні типу **`char`** оголошуються в такій самій манері, що й інші змінні. Ось кілька прикладів:

```
char response;
char itable, latan;
```

У цьому коді створюються три змінних типу **`char`**: **`response`**, **`itable`** і **`latan`**.

3.5.2.2. Символьні константи та ініціалізація

Припустимо, що необхідно ініціалізувати символьну константу буквою `'A'`. Комп'ютерні мови покликані полегшити вирішення цього завдання, тому не доведеться запам'ятовувати всі коди ASCII. Можна присвоїти символ `'A'` змінній **`grade`** за допомогою наступної ініціалізації:

```
char grade = 'A';
```

Одиночний символ, який обмежують одиночні лапки, являє собою символьну константу в *C*. Коли компілятор зустрічає конструкцію `'A'`, він перетворює її у відповідне кодове значення. Одиночні лапки тут є дуже важливими. Розглянемо ще один приклад:

```
char broiled;       // оголошення змінної типу char

broiled = 'T';      // правильно
broiled = T;        // неправильно!
broiled = "T";      // неправильно!
```

Якщо опустити лапки, то компілятор вважатиме, що **`T`** є ім'ям змінної. Якщо застосувати подвійні лапки, він сприйме `"T"` як рядок.

Оскільки символи насправді зберігаються у вигляді числових значень, для присвоювання значень можна також вказувати числові коди:

```
char grade = 65;    // правильно для ASCII,
                    // але стиль не дуже вдалий
```

В даному прикладі 65 має тип **`int`**, але оскільки це значення менше максимального значення типу **`char`**, воно може бути присвоєно змінній **`grade`** без будь-яких проблем. Оскільки 65 являє собою ASCII-код літери `'A'`, то в цьому прикладі змінній **`grade`** присвоюється значення `'A'`. Проте, слід звернути увагу на те, що у прикладі передбачається використання в системі кодування ASCII. Використання `'A'` замість 65 дає в результаті код, який працює в будь-якій системі. Таким чином, застосовувати символьні константи набагато краще, ніж значення числових кодів.

3.5.2.3. Друковані символи

Для вказівки на те, що необхідно вивести символ, у функції `printf()` використовується специфікатор %c. Символьна змінна зберігається як цілочислове значення та займає 1 байт. Отже, при виводі значення змінної типу **`char`** зі звичайним специфікатором %d буде отримано ціле число. Специфікатор формату %c повідомляє функції `printf()` про необхідність відобразити символ з кодовим значенням, що дорівнює цьому цілому числу.

Розглянемо код, в якому змінна **`char`** виводиться обома способами:

```
#include <stdio.h>
#include <windows.h>

int main(void)
{
    unsigned char ch;

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    printf("Введіть будь-який символ: ");
    scanf("%c", &ch);
    printf("Код символу %c = %d\n\n", ch, ch);

    for(int i = 32; i <= 255; i++)
    {
        if(i % 16 == 0)
            printf("\n");
        printf("%2c", i);
    }
    printf("\n");
    return 0;
}
```

Результат виконання програми наведено на рис. 3.7.

Функція `scanf()` зчитує символ, введений з клавіатури, а амперсанд (`'&'`) означає, що цей символ присвоюється змінній **`ch`**. Далі за допомогою функції `printf()` значення змінної **`ch`** виводиться два рази: спочатку як символ (специфікатор %c), а потім як десяткове ціле число (специфікатор %d). Зверніть увагу на те, що специфікатори функції `printf()` визначають спосіб відображення даних, але не те, як вони зберігаються в пам’яті (рис. 3.8).

У деяких реалізаціях *C* тип **`char`** є типом зі знаком. Це означає, що змінна

типу **char** може приймати значення з діапазону від –128 до 127. У інших реалізаціях тип **char** може бути беззнаковим і мати значення з діапазону від 0 до 255. В описі вашого компілятора має бути явно вказано, до якого різновиду належить тип **char**, або ви про це можете дізнатися, заглянувши у файл **limits.h**.

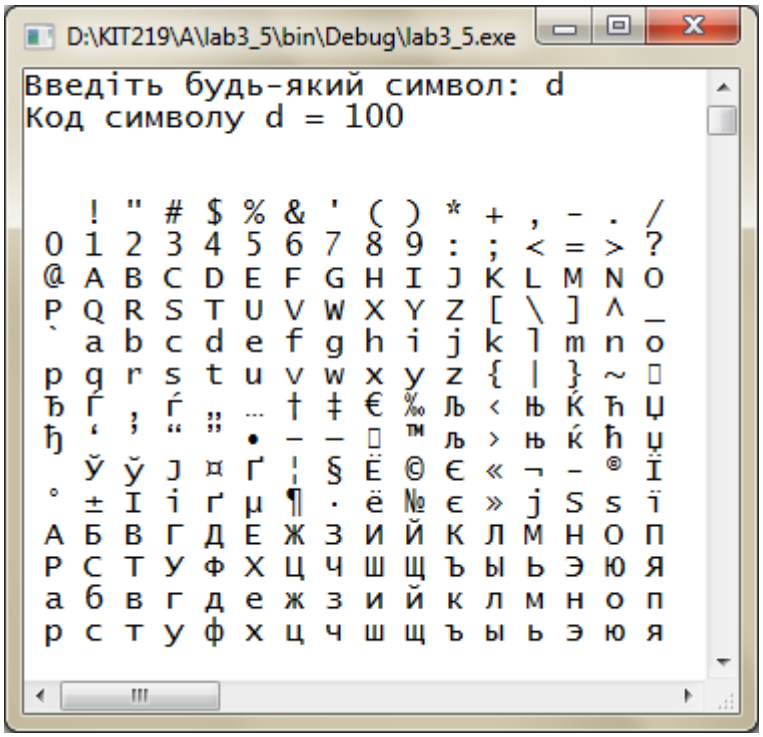


Рисунок 3.7 – Результат виконання програми для виводу на консоль будь-якого символу, а також таблиці символів з кодами від 32 до 255

Відповідно до стандарту С89, мова *С* дозволяє використовувати ключові слова **signed** і **unsigned** з типом **char**. Потім, незалежно від того, якими є дані типу **char** за замовчуванням, тип **signed char** буде зі знаком, а тип **unsigned char** – без знаку. Такі версії типу **char** є зручними, якщо цей тип застосовується для обробки невеликих цілих чисел. Для символів використовуйте стандартний тип **char** без модифікаторів.

3.5.3. Тип `_Bool`

Тип **_Bool**, що з’явився в стандарті С99, застосовується для представлення булевих значень, тобто логічних значень `true` (істина) і `false` (хибність). Оскільки в мові *С* для представлення `true` використовується значення 1, а для

представлення `false` – значення 0, – тип **`_Bool`** по суті є цілочисловим типом, який вимагає всього 1 біт пам’яті. Цього цілком достатньо, щоб охопити весь діапазон від 0 до 1.

У програмах булеві значення застосовуються при виборі того, який код повинен виконуватися наступним.

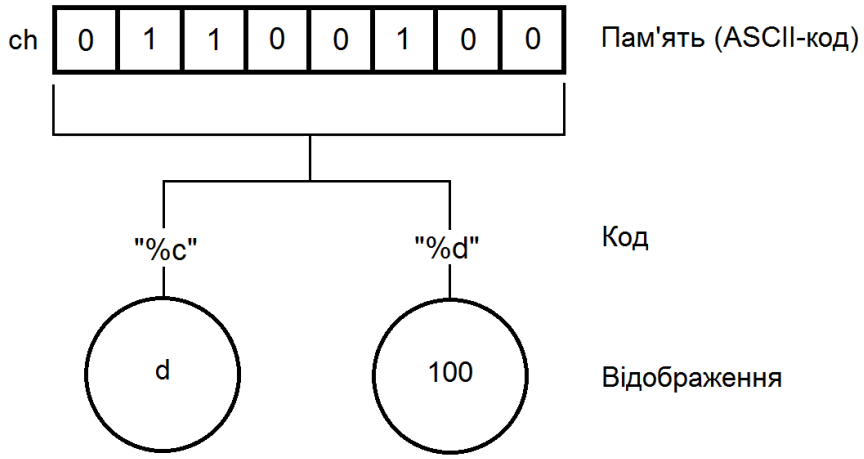


Рисунок 3.8 – Відображення символу 'd' на екрані та його зберігання у пам’яті

3.5.4. Портативні типи `stdint.h` і `inttypes.h`

Одне й те ж саме ім’я типу не обов’язково означає одне й те ж саме в різних системах. Було б чудово, якби мова *C* мала типи, зміст яких не залежав би від системи. Починаючи зі стандарту C99, щось подібне було досягнуто.

Це було зроблено в мові за рахунок створення додаткових імен для існуючих типів. Секрет в тому, що ці нові імена визначені в заголовному файлі **`stdint.h`**. Наприклад, **`int32_t`** представляє собою тип для 32-бітного цілого значення зі знаком. В системі, в якій використовується 32-бітний тип **`int`**, вказаний заголовок може визначати **`int32_t`** як псевдонім типу **`int`**. В іншій системі, де застосовується 16-бітний тип **`int`** і 32-бітний тип **`long`**, це ж саме ім’я (**`int32_t`**), може бути визначено як псевдонім для типу **`long`**. Тоді при створенні програми з використанням **`int32_t`** в якості типу та включенні заголовного файлу **`stdint.h`** компілятор буде замінювати тип **`int`** або **`long`** так, як це підходить для конкретної системи.

Розглянуті альтернативні імена є прикладами цілочислових типів з точною шириною. Тип **int32_t** містить в точності 32 біти. Не виключено, що цільова система не підтримує ці варіанти, тому цілочислові типи з точною шириною слід вважати необов'язковими.

Якщо система не здатна підтримувати типи з точною шириною, то стандарти C99 і C11 надають іншу категорію альтернативних імен, які є обов'язковими. Цей набір імен гарантує, що тип є досить великим щоб задовольняти специфікації, і немає інших типів, які можуть виконати потрібну роботу, але мають менший розмір. Ці типи називаються типами з мінімальною шириною. Наприклад, **int_least8_t** є псевдонімом найменшого доступного типу, який може зберігати 8-бітне цілочислове значення зі знаком. Якби найменший тип в конкретній системі був 16-бітним, то тип **int8_t** не визначався б. Однак тип **int_least8_t** був би доступним і, швидше за все, реалізованим як 16-бітний цілочисловий тип.

Звичайно, є програмісти, яких більше турбує швидкодія, ніж витрачання пам'яті. Для них стандарти C99 і C11 визначають набір типів, які забезпечать максимально швидкі обчислення. Ці типи називаються високошвидкісними типами з мінімальною шириною. Наприклад, тип **int_fast8_t** може бути визначений як альтернативне ім'я для цілочислового типу даних вашої системи, який забезпечує високошвидкісні обчислення за участю 8-бітних значень зі знаком.

Нарешті, деяких програмістів влаштовує тільки максимально можливий в системі цілочисловий тип. Такому типу відповідає ім'я **intmax_t** і він може зберігати будь-яке припустиме цілочислове значення зі знаком. Аналогічно, **uintmax_t** являє собою тип найбільшого допустимого цілочислового значення без знаку. До речі, зазначені типи можуть бути більшими ніж **long long** і **unsigned long**, оскільки реалізаціям *C* дозволено визначати типи, що виходять за рамки обов'язкових. Наприклад, деякі компілятори ввели тип **long long** ще до того, як він став частиною стандарту.

Стандарти C99 і C11 не тільки надають ці нові портативні імена типів,

але також сприяють вводу та виводу значень таких типів. Наприклад, функція `printf()` вимагає певних специфікаторів для конкретних типів.

Так що ж потрібно зробити, щоб відобразити значення **`int32_t`**, коли для одного визначення може вимагатися специфікатор %d, а для іншого – %ld? Поточний стандарт надає рядкові макроси, призначені для відображення таких типів. Наприклад, заголовний файл **`inttypes.h`** визначить `PRId32` як рядок, що представляє відповідний специфікатор (скажімо, `'d'` або `'l'`) для 32-бітного значення зі знаком.

Далі наведемо приклад, який ілюструє застосування такого типу і пов'язаного з ним специфікатора. Заголовок **`inttypes.h`** містить також і заголовний файл **`stdint.h`**, тому в програмі доведеться включати тільки файл **`inttypes.h`**. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <inttypes.h>  // підтримує портативні типи

int main(void)
{
    int32_t me32;       // me32 – це 32-бітна змінна зі знаком

    me32 = 45933945;

    SetConsoleOutputCP(1251);

    printf("Спочатку припустимо, що int32_t є int: ");
    printf("me32 = %d\n", me32);
    printf("Далі не будемо робити жодних припущень.\n");
    printf("Замість цього скористаємося \""
           "макросом\" з файлу inttypes.h: ");
    printf("me32 = %"PRId32"\n", me32);
    return 0;
}
```

Результат виконання програми наведено на рис. 3.9.

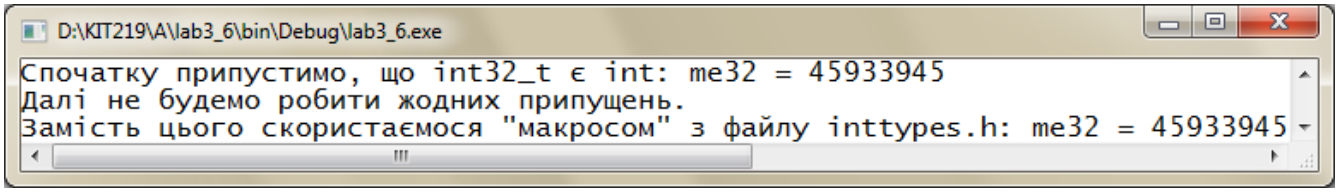


Рисунок 3.9 – Застосування в програмі типу int32_t

У фінальному виклику функції `printf()` аргумент `PRId32` замінюється своїм визначенням `"d"` з файлу **`inttypes.h`**, в результаті чого рядок приймає такий вигляд:

```
printf("me32 = %" "d" "\n", me32);
```

Однак *C* об'єднує рядки в подвійних лапках, які йдуть послідовно, в один рядок, даючи в результаті такий рядок:

```
printf("me32 = %d\n", me32);
```

Зверніть увагу, що в розглянутому прикладі також використовується керувальна послідовність (`\"`) для відображення подвійних лапок.

У даній лабораторній роботі не ставиться за мету вивчення всіх розширених цілочислових типів. Намір швидше полягає в демонстрації наявності цього рівня керування типами на той випадок, якщо це буде потрібно.

Незважаючи на те, що мова *C* перейшла на стандарт C11, навіть засоби стандарту C99 розробники компіляторів впроваджували в різному темпі та з різними пріоритетами. У зв'язку з цим, в деяких компіляторах може бути не реалізований заголовок **`inttypes.h`** і пов'язані з ним можливості.

3.5.5. Типи `float`, `double` і `long double`

Різноманітні цілочислові типи підходять для більшості проєктів з розробки програмного забезпечення. Проте програми, орієнтовані на математику та фінанси, часто оперують числами з рухомою комою. У мові *C* такі числа мають типи **`float`**, **`double`** або **`long double`**. Вони відповідають даним дійсного типу. Числа з рухомою комою дозволяють представляти набагато більший діапазон чисел, включаючи десяткові дроби. Представлення чисел з рухомою комою подібне до наукової форми запису, яка застосовується для відображення дуже великих і дуже маленьких чисел. Розглянемо таку форму запису.

У науковій формі запису числа представляються у вигляді десяткових чисел, помножених на степінь числа 10 (табл. 3.2).

У першому стовпці показана звичайна форма запису числа, у другому стовпці – наукова форма запису, а в третьому – експоненціальна форма, яка

представляє собою наукову форму запису, що зазвичай використовується при роботі з комп'ютерами. При цьому за позначенням **e** йде показник степені 10.

Таблиця 3.2 – Форми запису чисел

Число	Наукова форма запису	Експоненціальна форма запису
1 000 000 000	1.0×10^{9}	1.0e9
123 000	1.23×10^{5}	1.23e5
322,56	3.2256×10^{2}	3.2256e2
0,000056	5.6×10^{-5}	5.6e–5

Приклади чисел з рухомою комою: 2.58, 1.37e+7, 1.6E–19, 12E20.

Стандарт мови *C* вимагає, щоб тип **float** міг представляти мінімум шість значущих цифр і охоплював діапазон значень, щонайменше, від 10^{-37} до 10^{+37}. Перша вимога означає, що тип **float** повинен представляти, як мінімум, перші шість цифр такого числа, як 33.333333. Другу вимогу по достоїнству оцінять ті, хто оперує такими величинами, як маса Сонця (2.0e30 кілограм), електричний заряд протона (1.6e–19 кулона) або сума державного боргу. Часто для зберігання чисел з рухомою комою системи використовують 32 біти. Вісім бітів відводяться під значення експоненти та її знаку, а решта 24 біти служать для представлення неекспоненціальної частини числа, яка називається мантисою або значущою частиною числа, і її знаку.

Для представлення чисел з рухомою комою мова *C* пропонує також тип **double** (що забезпечує подвійну точність). Тип **double** має ті ж самі вимоги до мінімального діапазону можливих значень, що і **float**, але підтримує більш високу мінімальну кількість значущих цифр – 10. У типових представленнях типу **double** застосовуються 64 біти, а не 32. У деяких системах всі 32 додаткові біти використовуються для неекспоненціальної частини. Це призводить до збільшення кількості значущих цифр і скорочення помилок, пов'язаних з округленням. В інших системах частина цих бітів використовується для розміщення бі-

льшої експоненти, завдяки чому розширюється діапазон можливих значень. Будь-який з цих підходів забезпечує, як мінімум, 13 значущих цифр, що більш ніж задовольняє мінімальній вимозі стандарту.

Мова *C* допускає третій тип даних з рухомою комою: **long double**. Мета цього типу – досягнення більшої точності, ніж для типу **double**. Однак *C* гарантує тільки те, що точність типу **long double**, щонайменше, не поступається точності типу **double.**

3.5.5.1. Оголошення змінних з рухомою комою

Змінні з рухомою комою оголошуються та ініціалізуються аналогічним чином, як це було розглянуто раніше. Нижче наведено кілька прикладів:

```
float noah, jonah;
double trouble;
float planck = 6.63e-34;
long double gnp;
```

3.5.5.2. Вивід значень з рухомою комою

Функція `printf()` використовує специфікатор формату %f для виводу чисел типу **float** і **double** в десятковому представленні та специфікатор %e для виводу чисел в експоненціальному представленні. Якщо ваша система підтримує шістнадцятковий формат чисел з рухомою комою, то замість `'e'` або `'E'` можна застосовувати `'a'` чи `'A'`. Для виводу даних типу **long double** потрібні специфікатори %Lf, %Le і %La. Зверніть увагу, що для виводу як **float**, так і **double** використовується специфікатор %f, %e або %a. Причина полягає в тому, що мова *C* автоматично розширює значення **float** до типу **double**, коли вони передаються в якості аргументів будь-якій функції, наприклад, такій як `printf()`, в прототипі якої тип аргументу не визначений явно. Ця поведінка демонструється в наступній програмі:

```
#include <stdio.h>
#include <windows.h>

int main(void)
{
    float aboat = 32000.0;
    double abet = 2.14e9;
```

```
    long double dip = 5.32e-5;

    SetConsoleOutputCP(1251);
    printf("%f може бути записано як %e\n", aboat, aboat);
    // для виводу наступного рядка потрібен компілятор,
    // який підтримує C99 або C11
    printf("Та його %a в шістнадцятковій формі запису\n",
           aboat);
    printf("%f може бути записано як %e\n", abet, abet);
    __mingw_printf("%lf може бути записано як %le\n",
                   dip, dip);
    return 0;
}
```

Результат виконання програми наведено на рис. 3.10.

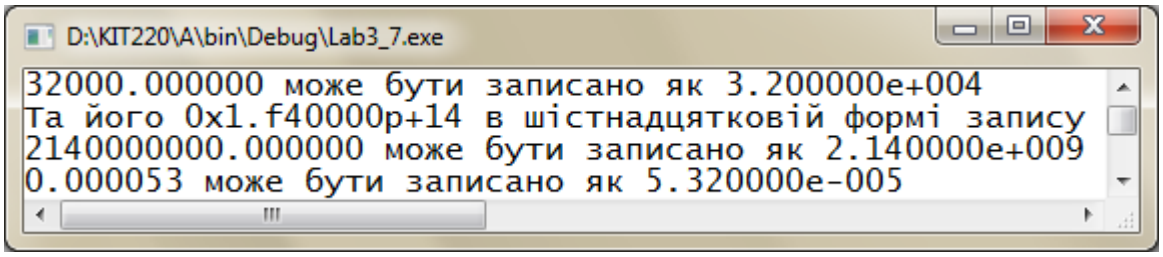


Рисунок 3.10 – Виводимо на консоль даних типів float, double і long double

Цей приклад ілюструє стандартний вивід на консоль. Необхідно відзначити, що для правильного виводу значення типу **`long double`**, необхідно використовувати функцію `__mingw_printf()` або приводити змінну **`dip`** до типу **`double`**.

3.5.5.3. Переповнення та втрата значущості

Припустимо, що найбільше можливе значення типу **`float`** дорівнює приблизно 3.4E38, і потрібно виконати наступну програму:

```
#include <stdio.h>
#include <windows.h>

int main(void)
{
    float toobig = 3.4E38 * 100.0f;

    SetConsoleOutputCP(1251);
    printf("Виводимо на консоль дуже велике значення\n");
    printf("%e\n", toobig);
    return 0;
}
```

Це приклад переповнення, коли в результаті обчислень виходить занадто велике число, щоб його можна було б правильно представити. Поведінка системи в таких випадках зазвичай є невизначеною, але в ситуації, що розглядається змінній **`toobig`** присвоюється спеціальне значення, яке позначає нескінченність, і функція `printf()` відображає або `inf`, або `infinity` (рис. 3.11).

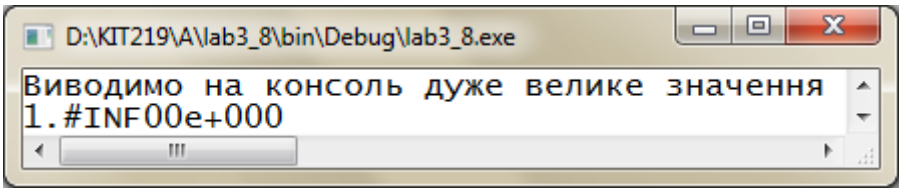


Рисунок 3.11 – Виводимо на консоль дуже велике значення для типу float

При діленні дуже малих чисел ситуація буде більш складною. Число типу **`float`** зберігається у вигляді поєднання показника степені та значущої частини, або мантиси. В даному випадку це число, яке має мінімально можливий показник степеня, а також найменше значення, яке використовує всі доступні біти, відведені для представлення мантиси. Це буде найменше число, представлене з найбільшою точністю, доступною для типу **`float`**. Тепер розділимо його на 2. Зазвичай це призводить до зменшення показника степеня, але в даному випадку показник вже досяг своєї нижньої межі. Відбувається зсув бітів мантиси вправо, із звільненням першої позиції та втратою останньої двійкової цифри. Аналогічна картина виникає, якщо вибрати 10 в якості основи системи числення, взяти число з чотирма значущими цифрами, наприклад 0.1234E–10, і поділити його на 10, отримавши в підсумку 0.0123E–10. Ви отримаєте результат, але в процесі ділення втратите цифру. Ця ситуація називається втратою значущості, а значення з рухомою комою, які втратили повну точність типу, в мові *C* називаються субнормальними. Таким чином, ділення найменшого додатного значення з рухомою комою на 2 дає субнормальне значення. Ділення на досить велике значення призведе до втрати всіх цифр, і ви отримаєте в результаті 0. На даний час бібліотека *C* надає функції, які дозволяють перевірити, чи не призведуть обчислення до субнормальних значень.

Існує ще одне спеціальне значення з рухомою комою: `NaN` (not-a-number – не є числом). Наприклад, ви передаєте функції `asin()` деяке значення, а вона повертає кут, для якого передане значення є синусом. Однак значення синуса не може бути більше 1. Тому функція не визначена для значень, що перевищують 1. У таких випадках функція повертає значення `NaN`, яке функція `printf()` відображає у вигляді `nan`, `NaN` або якимось схожим чином.

3.5.6. Комплексні та уявні типи

У багатьох наукових і технічних розрахунках використовуються комплексні та уявні числа. Стандарт С99 підтримує ці числа, але з деякими застереженнями. В автономній реалізації, такій як застосовується для вбудованих процесорів, необхідність в наявності цих типів відсутня. Також в загальному випадку уявні типи є необов'язковими. У стандарті С11 весь пакет підтримки комплексних чисел зроблений необов'язковим.

Існують три комплексних типи: **`float _Complex`**, **`double _Complex`** і **`long double _Complex`**. Змінна типу **`float _Complex`**, наприклад, буде містити два значення **`float`**, одне з яких представляє дійсну частину комплексного числа, а інше – його уявну частину. Аналогічно, існують три уявних типи: **`float _Imaginary`**, **`double _Imaginary`** і **`long double _Imaginary`**.

Включення заголовного файлу **`complex.h`** уможливлює підстановку слова **`complex`** замість **`_Complex`** і слова **`imaginary`** замість **`_Imaginary`**, а також застосування символу **`I`** для добування квадратного кореня з –1.

У стандарті мови *C* не ввели **`complex`** у якості ключового слова. Комітет зі стандартизації зазвичай не поспішає вводити нове ключове слово, оскільки це може призвести до втрати допустимості існуючого коду, в якому таке слово застосовувалося в якості ідентифікатора.

Наприклад, до появи стандарту С99 багато програмістів вже використовували **`struct complex`** для представлення комплексних чисел або, можливо, психологічних станів. Перетворення слова «complex» на ключове слово перетворило б попередні випадки його застосування на синтаксичні помилки. З ін-

шого боку, використання поєднання **`struct _Complex`** є значно менш імовірним, особливо з урахуванням того, що ідентифікатори з початковим символом підкреслення вважаються зарезервованими.

Таким чином, комітет зупинився на **`_Complex`** в якості ключового слова та зробив варіант **`complex`** доступним для тих, кому не потрібно турбуватися з приводу конфліктів з попередніми застосуваннями.

3.6. Розміри типів

Які розміри типів використовуються у вашій системі? Щоб з'ясувати це, спробуйте виконати наступну програму:

```
#include <stdio.h>
#include <windows.h>
int main(void)
{
    SetConsoleOutputCP(1251);
    printf("Тип int         має розмір %2d байт.\n",
           sizeof(int));
    printf("Тип char        має розмір %2d байт.\n",
           sizeof(char));
    printf("Тип long        має розмір %2d байт.\n",
           sizeof(long));
    printf("Тип long long   має розмір %2d байт.\n",
           sizeof(long long));
    printf("Тип double      має розмір %2d байт.\n",
           sizeof(double));
    printf("Тип long double має розмір %2d байт.\n",
           sizeof(long double));
    return 0;
}
```

Результат виконання програми наведено на рис. 3.12.

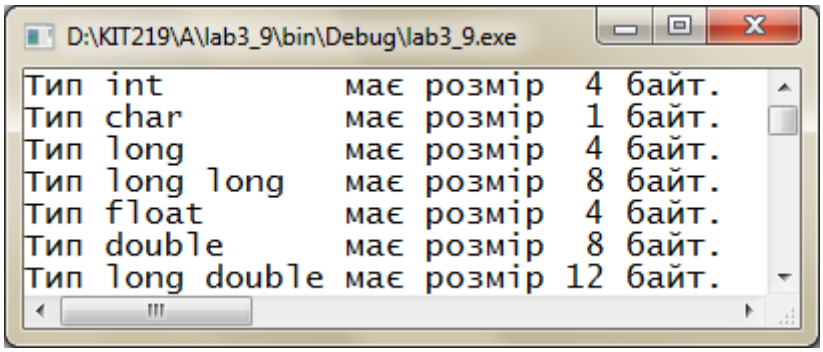


Рисунок 3.12 – Виводимо на консоль розміри різних типів у байтах

У мові *C* є вбудована операція **sizeof**, яка повертає розмір типу в байтах. Для такого застосування **sizeof** в стандартах C99 і C11 надається специфікатор %zd. Компілятори, які є несумісними з цими стандартами, можуть вимагати замість нього специфікатор %u або %lu.

Розглянута програма знаходить розміри тільки шести типів, але ви легко можете її модифікувати, щоб вона визначала розмір будь-якого іншого типу, який вас цікавить. Зверніть увагу, що розміром типу **char** обов'язково буде 1 байт, тому що в мові *C* розмір одного байта визначається в термінах **char**. Таким чином, в системі з 16-бітним типом **char** і 64-бітним **double** операція **sizeof** повідомить, що тип **double** має розмір 4 байти. Для отримання більш докладної інформації про граничні розміри типів можна переглянути заголовні файли **limits.h** і **float.h**.

До речі, зверніть увагу на те, що в кількох останніх рядках оператор `printf()` розміщається на двох рядках. Його можна розділяти на більшу кількість частин за умови, що розрив не відбувається всередині розділу, розміщеного між подвійними лапками, або всередині слова.

Хід виконання роботи:

1. Опрацюйте теоретичний матеріал даної лабораторної роботи.

2. Модифікуйте тексти *C*-програм шляхом внесення до них власних змін. Перевірте отримані програми на працездатність. Помістіть до звіту умови завдань, тексти програм і копії екранів консолі, де в заголовку кожного вікна повинно бути вказане ваше прізвище англійською мовою.

3. Відповідно до вашого варіанта наведіть докладну інформацію про кожен з типів даних, які відповідають вашому варіанта (табл. 3.3).

4. Напишіть *C*-програму, яка б обчислювала об'єм пам'яті, що займає кожен з цих типів (див. табл. 3.3). Порівняйте їх між собою з точки зору об'єму пам'яті, яку вони займають.

Таблиця 3.3 – Типи даних, інформацію про які треба навести

№ варіанта	Перший тип даних	Другий тип даних
1	`short`	`unsigned long int`
2	`signed int`	`float`
3	`short int`	`signed long long`
4	`unsigned`	`long long`
5	`signed short int`	`unsigned long long int`
6	`long`	`double`
7	`signed short`	`unsigned long long`
8	`long int`	`long double`
9	`unsigned short`	`float`
10	`unsigned int`	`double`
11	`unsigned short int`	`long double`
12	`signed long`	`long long int`
13	`int`	`signed long long int`
14	`signed long int`	`long double`
15	`signed int`	`float`
16	`signed`	`unsigned long`
17	`float`	`short int`
18	`double`	`signed long`
19	`unsigned long long`	`long double`
20	`signed long long int`	`float`

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:
   – умови виконання завдань і модифіковані тексти *C*–програм разом з результатами;
   – докладний опис кожного з заданих типів даних та їх порівняння;

– текст програми для обчислення об’єму пам’яті, що займає кожний з типів;

– результат виконання *C*-програми для обчислення об’єму пам’яті.

4. Висновки по роботі.

Контрольні запитання:

1. Які базові типи даних існують у мові С?

2. Які особливості використання цілочислових типів даних?

3. Як у програмі описати змінну з іменем **`x`** цілого типу?

4. Як у змінну цілого типу записати число?

5. Які особливості мають типи даних з рухомою комою?

6. Як описати змінну, що приймає значення з рухомою комою?

7. Як у змінну з рухомою комою записати числові значення?

8. Як привести змінну типу **`float`** до типу **`int`**?

9. Як привести змінну типу **`int`** до типу **`double`**?

10. Які особливості використання даних типу **`char`** (символьних даних) у програмі на *C*?

11. Які особливості використання даних типу **`_Bool`** (логічний тип)?

12. Як визначити розмір того чи іншого типу в С?

13. Яким чином здійснюється ініціалізація змінних різних типів?

14. Яким чином визначити максимально допустиме (мінімально допустиме) значення змінної певного типу?

15. Які особливості використання типу **`enum`**?

16. Які особливості застосування типу **`void`** у програмах на С?

17. Чи можна оголошувати змінну типу **`void`** у програмі?

ЛАБОРАТОРНА РОБОТА №4
ДОСЛІДЖЕННЯ ІНСТРУКЦІЙ ВИБОРУ

Мета роботи: ознайомитися з інструкціями вибору, які використовуються у мові *C*; навчитися застосовувати інструкції **`if`** та **`if...else`**, а також вкладені інструкції **`if...else`**; розглянути ситуації застосування умовного оператора (**`?:`**); навчитися застосовувати інструкцію множинного вибору **`switch`**.

Теоретична частина

4.1. Інструкції вибору

У мові програмування С синтаксис інструкції вибору **`if...else`** виглядає наступним чином [4–8]:

```
if(вираз)
    інструкція;
else
    інструкція;
```

Як і в інших мовах програмування, гілка **`else`** не є обов'язковою.

У простих логічних виразах використовуються наступні знаки логічних операцій: **`>, <, >=, <=, ==, !=`**. У випадку, коли тіло тієї або іншої гілки складається з декількох закінчених виразів, що поділяються крапкою з комою, воно розміщується в фігурних дужках:

```
if(вираз)
{
    інструкція_1;
    інструкція_2;
    ...
}
else
{
    інструкція_N;
    ...
}
```

В С можна використовувати вкладені інструкції **`if...else`**. При цьому рекомендується вкладати у зовнішню гілку **`else`**, а не в **`if`**, оскільки це дозволяє уникати неоднозначності тлумачення інструкції.

Подивіться на таку конструкцію:

```
if(...)
    if(...)
        ...;
    else
        ...
else
    if(...)
        ...;
    else
        ...;
```

Для більш легкого сприйняття людиною тексту програми, відступами підкреслено, що і куди вкладено. Однак для компілятора мови С відступи ніякої ролі не грають, і приналежність першої гілки **else** не є очевидною. Її можна було б помилково віднести до першого **if**, в результаті чого другий **else** був би взагалі недоречним, оскільки він не належав би до жодного **if**. В даному випадку такої помилки не буде, тому що компілятор керується таким правилом: гілка **else** відноситься до найближчого до неї зверху **if**, у якого ще немає гілки **else**. Саме тому перший **else** відноситься до другого **if** (оскільки він ближче до нього), а другий **else** до першого **if**, тому що другий **if** вже «зайнятий» попереднім **else**.

Тепер поглянемо на наступну конструкцію:

```
if(...)
    if(...)
        ...;
else
    if(...)
        ...;
    else
        ...;
```

Програміст відступами показав, що він хоче, щоб перший **else** відносився до зовнішнього **if**. Однак компілятор, керуючись правилом віднесення гілок **else**, розуміє програму наступним чином:

```
if(...)
    if(...)
        ...;
    else
        if(...)
```

```
        ...;
    else
        ...;
```

При цьому програма буде робити не те, що хотів програміст: якщо у виразі при першому **if** буде повертатися хибність, то жоден з **else** просто не спрацює. Однак в таких складних випадках є вихід – це використання фігурних дужок, навіть якщо тіло умовної інструкції складається усього з одного закінченого виразу:

```
if(...)
{
    if(...)
        ...;
}
else
    if(...)
        ...;
    else
        ...;
```

В такому випадку програма буде сприйматися компілятором саме так, як задумав програміст. Частіше за все на практиці використовується подібна конструкція множинного вибору:

```
if(...)
{
    ...
}
else if(...)
{
    ...
}
else if(...)
{
    ...
}
else
{
    ...
}
```

В даному випадку, з точки зору компілятора, кожний **else** відноситься до попереднього **if**, але вирівнювання робиться таким, начебто в *C* є умовний оператор множинного вибору. Результат роботи конструкції є таким, що вико-

нується хоча б одна гілка. Як тільки вона виконається, вся конструкція завершує роботу.

4.2. Умовний вираз

У мові програмування С існує скорочений запис інструкції **`if...else`** у вигляді умовного виразу, результат якого може бути присвоєний змінній:

```
(логіч_вираз) ? вираз_1 : вираз_2
```

Цей запис треба розуміти наступним чином: якщо `логіч_вираз` повертає істину, то увесь вираз повертає `вираз_1`; якщо `логіч_вираз` повертає хибність, то увесь вираз повертає `вираз_2`.

Наприклад:

```
x = 13;
y = 17;
z = (x > y) ? x - 1 : y - 1;
```

В даному випадку **`z`** отримає значення 16. Такий умовний вираз є дуже зручним, однак сфера його застосування обмежена простими випадками використання.

4.3. Логічні операції "І" (**`&&`**) та "АБО" (**`||`**)

Як відомо логічний вираз може бути не тільки простим, але й складеним. Логічні операції "І" та "АБО" у мові програмування С позначаються відповідно парними знаками амперсанда (**`&&`**) та вертикальної риски (**`||`**). Їх пріоритет менший, ніж у простих логічних операцій, тому прості логічні вирази при їх об’єднанні у складені логічні вирази можна не поміщати в дужки.

Приклад складеного логічного виразу:

```
a > 100 && b != 0
```

4.4. Оператор **`switch`**

При організації множинного вибору, коли перевіряється значення змінної на відповідність тому або іншому значенню, буває зручно використовувати не

умовну інструкцію **if...else**, а інструкцію множинного вибору **switch**. Її синтаксис можна записати наступним чином:

```
switch(цілочислова_змінна)
{
    case константа1:
        інструкція;
    case константа2:
        інструкція;
        ...
    default:
        інструкція;
}
```

В дужках після слова **switch** може стояти не тільки змінна, але й вираз, результат виконання якого повертає ціле значення (може бути й символ). Константи поруч з **case** також можуть бути результатом виконання виразів. Константи можна групувати в одному **case** (наприклад, **case** 12: **case** 13: **case** 18: інструкція;). Гілка **default** є необов'язковою.

При виконанні інструкції **switch**, задане значення в круглих дужках порівнюється з константами. Як тільки збіг значень буде знайдений, всі подальші вкладення в усіх **case** інструкціях почнуть виконуватися. Іншими словами, виконується не тільки **case**, де відбувся збіг, але й усі гілки **case** , які розташовані нижче (і гілка **default** також), константи яких не збігаються зі значенням при **switch**. Наприклад, в результаті виконання цієї програми

```
int a = 1;

switch(a)
{
     case 0: printf("%d ",  0);
     case 1: printf("%d ",  1);
     case 2: printf("%d ",  2);
    default: printf("%d ", -1);
}
printf("\n");
```

на екрані консолі буде зазначено: 1 2 –1, оскільки як тільки збіг був виявлений, усі інструкції, які знаходилися нижче, були виконані.

Щоб цього не відбулося, наприкінці інструкцій, що належать до певного ключового слова **case**, дописують інструкцію **break**, яка здійснює примусовий вихід з усієї конструкції (в даному випадку **switch**).

Наприклад фрагмент програми

```
int a = 1;
switch(a)
{
    case 0: printf("%d\n",  0);
            break;
    case 1: printf("%d\n",  1);
            break;
    case 2: printf("%d\n",  2);
            break;
   default: printf("%d\n", -1);
}
```

виведе тільки одиницю, оскільки виконання усієї інструкції **switch** припиниться після виконання інструкції **break** при **case** 1.

Розглянемо приклад застосування інструкції множинного вибору **switch**. Текст програми для обчислення простого арифметичного виразу має такий вигляд:

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

int main(void)
{
    float a, b, c;
    char op;

    SetConsoleOutputCP(1251);

    puts("================================================");
    puts(" Програма обчислює значення арифметичного виразу ");
    puts("================================================");
    puts(" Можна використовувати операції: +, -, *, /      ");
    puts(" Формат вводу: оп_1 операція оп_2 (Enter)        ");
    puts("================================================");
    printf(" Введіть вираз: ");
    scanf("%f %c %f", &a, &op, &b);

    switch(op)
    {
        case '+': c = a + b;
                  break;
```

```
            case '-': c = a - b;
                      break;
            case '/': if(b != 0.0)
                          c = a / b;
                      else
                      {
                          printf("Помилка: ділення на нуль");
                          exit(1);
                      }
                      break;
            case '*': c = a * b;
                      break;
             default: printf("Немає визначених операцій");
                      exit(1);
        }
        printf("\n %.6f %c %.6f = %.6f\n", a, op, b, c);
        return 0;
    }
```

Результат роботи програми для обчислення простого арифметичного виразу із застосуванням операції `'+'`, наведено на рис. 4.1.

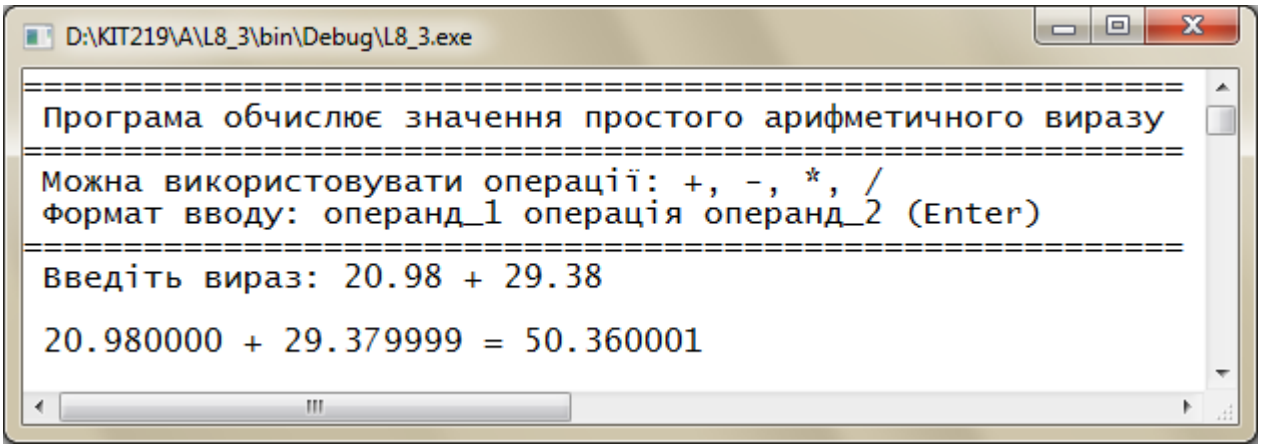


Рисунок 4.1 – Результат роботи програми для обчислення простого арифметичного виразу із застосуванням операції '+'

Функція `exit()` призводить до негайного закінчення роботи програми і повертає керування до операційної системи. Для використання цієї функції в програмі необхідно підключити заголовний файл **`stdlib.h`**.

Хід виконання роботи:

1. Опрацюйте теоретичний матеріал даної лабораторної роботи.

2. Напишіть текст *C*-програми, яка б вводила два операнди **a** і **b** та один з двох можливих символів арифметичної операції, і в залежності від цієї операції обчислювала б значення виразу. Для обчислення виразу необхідно застосувати умовний оператор (`?:`). Наведіть текст *C*-програми та результати її роботи для обох операцій.

Приклади зовнішнього вигляду консолі для операцій `'+'` і `'*'` наведені на рис. 4.2 і рис. 4.3.

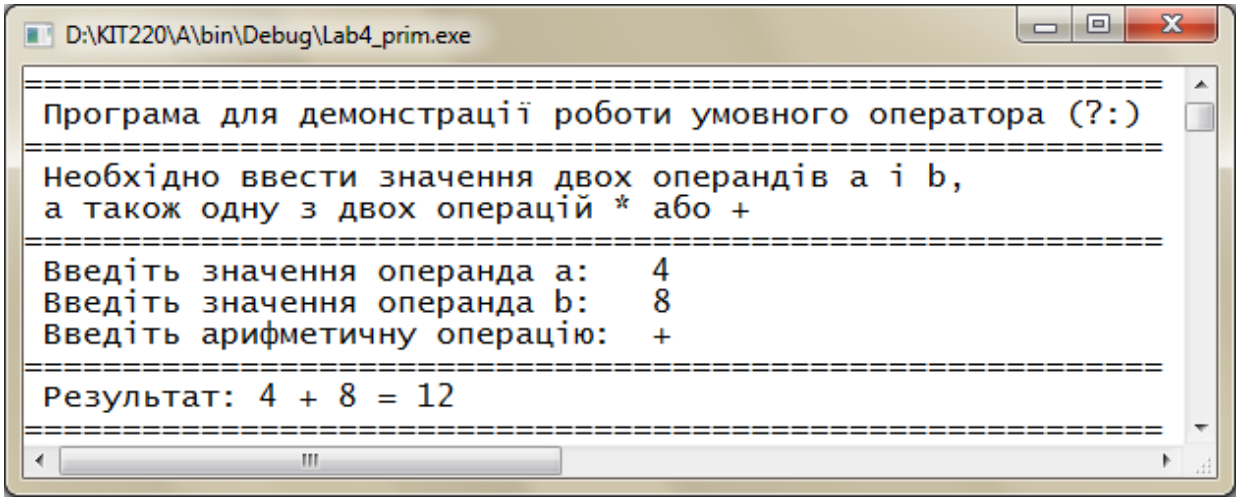


Рисунок 4.2 – Результат роботи програми для операції '+'

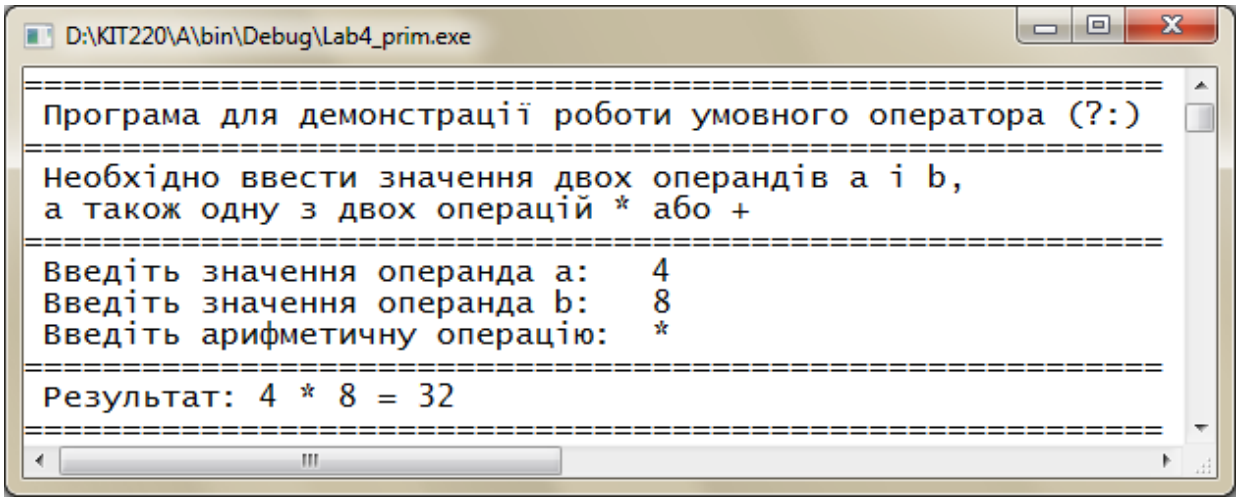


Рисунок 4.3 – Результат роботи програми для операції '*'

Перш ніж вводити символ операції, необхідно застосувати функцію `rewind(stdin)`, яка допоможе очистити буфер вводу для коректної роботи програми. Значення операндів, їх тип, а також дві арифметичні операції для вашого варіанта наведені в табл. 4.1.

Таблиця 4.1 – Дані для виконання завдання 2

Варіант	**a**	**b**	Тип операндів	Операції
1	5.36	6.27	**float**	*, –
2	243	332	**int**	%, +
3	85	36	**unsigned int**	^, ¦
4	25.36	12.29	**double**	/, *
5	41.08	29.87	**float**	+, –
6	23	12	**short int**	+, %
7	71	10	**int**	&, +
8	19.20	92.76	**double**	/, –
9	4.09	16.11	**float**	+, –
10	36	89	**int**	+, %
11	177	398	**unsigned int**	¦, &
12	70.92	38.48	**double**	/, *
13	10.93	37.81	**float**	+, /
14	20	83	**short int**	%, +
15	25	876	**int**	*, %
16	85.99	11.93	**double**	–, +
17	5.18	3.88	**float**	*, –
18	111	243	**int**	%, +
19	73	46	**unsigned int**	^, ¦
20	15.72	24.15	**double**	/, *

3. Відповідно до вашого варіанта (табл. 4.2), напишіть *C*-програму для обчислення значення функції. В програмі необхідно застосувати вкладену інструкцію **`if...else`** і навести текст програми та копії екранів для усіх трьох гілок алгоритму. Схему алгоритму наводити не треба.

4. Для програми обчислення простого арифметичного виразу, яка розміщена наприкінці теоретичної частини даної лабораторної роботи, наведіть копії екранів для усіх гілок алгоритму (шість копій екранів). Розберіться у принципі роботи інструкції **`switch`** і будьте готові до пояснення отриманого результату за будь-якою з гілок алгоритму.

Таблиця 4.2 – Дані для виконання завдання 3

Варіант	Формула
1	$f(p,q)=\begin{cases}\dfrac{p+q}{2q}, \text{ якщо } q \geq 100;\\ p-2q, \text{ якщо } 0<q<100,\ p<20;\\ p-3q, \text{ в інших випадках.}\end{cases}$
2	$f(x,y)=\begin{cases}x+y, \text{ якщо } x>0;\\ xy, \text{ якщо } x \leq 0,\ y<0;\\ 5x, \text{ в інших випадках.}\end{cases}$
3	$f(a,b)=\begin{cases}3a^2, \text{ якщо } a>5;\\ a/b, \text{ якщо } 0<a \leq 5,\ b \neq 0;\\ b+a-1, \text{ в інших випадках.}\end{cases}$
4	$f(m,n)=\begin{cases}m+n^2, \text{ якщо } n<0,\ m>0;\\ m+2n, \text{ якщо } m \leq 0,\ n<0;\\ m+1, \text{ в інших випадках.}\end{cases}$
5	$f(x,y)=\begin{cases}x+y, \text{ якщо } x>0, y<0;\\ x/(y-1), \text{ якщо } x>0,\ y>1;\\ x-y, \text{ в інших випадках.}\end{cases}$
6	$f(k,m)=\begin{cases}m/k, \text{ якщо } k>0;\\ \lvert m+k\rvert, \text{ якщо } k \leq 0,\ m<0;\\ km, \text{ в інших випадках.}\end{cases}$
7	$f(a,b)=\begin{cases}a-b, \text{ якщо } a>10;\\ \dfrac{a-b}{a+b}, \text{ якщо } 0<a \leq 10,\ b>0;\\ b, \text{ в інших випадках.}\end{cases}$
8	$f(x,y)=\begin{cases}x-\lvert y\rvert, \text{ якщо } x>0;\\ y^2-x, \text{ якщо } x \leq 0,\ y<0;\\ y+1, \text{ в інших випадках.}\end{cases}$
9	$f(a,b)=\begin{cases}3a+2, \text{ якщо } a<0,\ b=0;\\ 3a/2b, \text{ якщо } a<0,\ b \neq 0;\\ 3a-2b, \text{ в інших випадках.}\end{cases}$
10	$f(t,s)=\begin{cases}3t-1, \text{ якщо } s>4;\\ \sqrt{\lvert t-s\rvert}, \text{ якщо } s \leq 4,\ t \neq 0;\\ s+2, \text{ в інших випадках.}\end{cases}$

Продовження таблиці 4.2

Варіант	Формула
11	$f(m,n)=\begin{cases}\dfrac{2m}{n-3}, \text{ якщо } n\neq 3,\ m\leq 3;\\ \sqrt{m-3}-n, \text{ якщо } 3<m;\\ m+n, \text{ в інших випадках.}\end{cases}$
12	$f(x,y)=\begin{cases}y-x, \text{ якщо } x<0;\\ y^2-x, \text{ якщо } x\geq 0,\ y<0;\\ sin\, y, \text{ в інших випадках.}\end{cases}$
13	$f(x,y)=\begin{cases}y/x, \text{ якщо } x\neq 0;\\ y^2, \text{ якщо } x=0,\ y<0;\\ 5, \text{ в інших випадках.}\end{cases}$
14	$f(t,s)=\begin{cases}\sqrt[4]{t-s}, \text{ якщо } t\geq s,\ 2<s\leq 4;\\ s^4+2t, \text{ якщо } t<0;\\ t+2, \text{ в інших випадках.}\end{cases}$
15	$f(m,n)=\begin{cases}5/m-n/5, \text{ якщо } n>-5,\ m\neq 0;\\ 3m+n^2, \text{ якщо } n\leq -5;\\ 2mn, \text{ в інших випадках.}\end{cases}$
16	$f(x,y)=\begin{cases}\dfrac{x-2}{y-4}, \text{ якщо } y\neq 4,\ -y<x\leq y;\\ y/x, \text{ якщо } x\neq 0,\ y<x;\\ yx, \text{ в інших випадках.}\end{cases}$
17	$f(a,b)=\begin{cases}a-2b, \text{ якщо } a>b,\ b\geq 0;\\ \lvert a\rvert, \text{ якщо } a<0,\ b\neq 0;\\ -b, \text{ в інших випадках.}\end{cases}$
18	$f(x,y)=\begin{cases}2x-y, \text{ якщо } 0\leq x<5;\\ x^2, \text{ якщо } x\leq 0,\ y<0;\\ 5y+1, \text{ в інших випадках.}\end{cases}$
19	$f(x,y)=\begin{cases}x+3y, \text{ якщо } \lvert x-y\rvert>2;\\ 3, \text{ якщо } x=y;\\ 5x/y, \text{ в інших випадках.}\end{cases}$
20	$f(k,m)=\begin{cases}km-4, \text{ якщо } k>m/2;\\ k-4m, \text{ якщо } m>k/2;\\ m-4, \text{ в інших випадках.}\end{cases}$

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.

2. Мету роботи.

3. Результати виконання роботи:

– тексти трьох *С*-програм;

– результати виконання *С*-програм (11 копій екрану).

4. Висновки по роботі.

Контрольні запитання:

1. Чим відрізняється умовний вираз від умовної інструкції?

2. Який вигляд має умовний оператор?

3. Що таке повна та неповна форми умовної інструкції?

4. Вирази якого типу можуть визначати умови?

5. Які значення виразу, що визначають умову, вважаються істинними, а які хибними?

6. Які операції відносяться до операцій відношення?

7. Чим відрізняється операція `"=="` від операції `"="`?

8. Які операції відносяться до логічних? Який вони мають пріоритет?

9. Чому може дорівнювати значення виразу відношення або логічного виразу?

10. Коли застосовується вкладення умовних інструкцій?

11. Що собою являє інструкція **`switch`**? Як нею користуватися?

12. Як записати інструкцію **`switch`** за допомогою умовних інструкцій?

ЛАБОРАТОРНА РОБОТА №5
ДОСЛІДЖЕННЯ МЕТОДІВ СОРТУВАННЯ ОДНОВИМІРНОГО МАСИВУ

Мета роботи: вивчити питання сортування одновимірних цілочислових масивів; розглянути класифікацію алгоритмів сортування одновимірних масивів; дослідити прості алгоритми сортування та порівняти їх між собою; реалізувати алгоритми сортування за допомогою *C*-програм.

Теоретична частина

5.1. Сортування

Під час виконання даної лабораторної роботи для кожного завдання необхідно написати *C*-програму, яка здійснює генерування випадкових цілих чисел (в залежності від завдання) для одновимірного масиву та здійснює їх сортування відповідно до зазначеного алгоритму, а також виводить на консоль всі кроки сортування. Обмеженням на вхідні дані є визначений діапазон чисел і максимально допустимий розмір одновимірного масиву.

Задача сортування є базовою задачею, так само як і задача пошуку. У практичних умовах між цими задачами існує тісний взаємозв’язок. Вирішенню проблем, які пов’язані з сортуванням, присвячена велика кількість фундаментальних наукових досліджень, розроблено безліч алгоритмів [9].

У загальному розумінні сортування слід розглядати як процес перегрупування заданої множини об’єктів у певному порядку. Часто при сортуванні великих об’ємів даних недоцільно переставляти самі елементи, тому для вирішення задачі проводиться упорядкування елементів за індексами. Тобто індекси елементів вибудовують у такій послідовності, що відповідні їм значення елементів виявляються відсортованими за умовою задачі.

Сортування застосовується для полегшення пошуку елементів в упорядкованій множині. Завдання сортування елементів масиву є одним з фундаментальних завдань у програмуванні.

Сортування – це впорядкування набору однотипних даних за зростанням або за зменшенням.

Найчастіше при сортуванні даних лише частина їх використовується в якості ключа сортування.

Ключ сортування – це частина даних, яка визначає порядок елементів.

Таким чином, ключ бере участь у порівняннях, але при обміні елементів відбувається переміщення всієї структури даних. Наприклад, у списку поштової розсилки в якості ключа може використовуватися поштовий індекс, але сортується вся адреса. При вирішенні завдань сортування масивів ключ і дані збігаються.

Для того, щоб впорядкувати дані, можна застосовувати стандартну функцію `qsort()`, що входить до стандартної бібліотеки *C*. Однак різні підходи до сортування мають різні характеристики. Незважаючи на те, що деякі способи сортування можуть бути в середньому кращими за інші, жоден з алгоритмів не є ідеальним для всіх випадків.

Використання функції `qsort()` не є універсальним рішенням для всіх завдань сортування.

По-перше, функцію загального призначення, таку як `qsort()`, неможливо застосувати в усіх ситуаціях. Наприклад, дана функція сортує тільки масиви в пам’яті й не може сортувати дані, що зберігаються у зв’язних списках.

По-друге, `qsort()` – це параметризована функція, завдяки чому вона може обробляти широкий набір типів даних, але внаслідок цього працює повільніше, ніж еквівалентна функція, що розрахована на будь-який один тип даних.

По-третє, алгоритм швидкого сортування, застосований у функції `qsort()`, може виявитися не самим ефективним в деяких конкретних ситуаціях.

5.2. Оцінювання алгоритмів сортування

Існує велика кількість різних алгоритмів сортування. Всі вони мають свої позитивні та негативні сторони. Розглянемо загальні критерії оцінки алгоритмів сортування:

1) швидкість роботи алгоритму сортування. Вона безпосередньо пов’язана з кількістю порівнянь і кількістю обмінів, що відбуваються під час сортування, причому обміни займають більше часу. Порівняння відбувається тоді, коли один елемент масиву зіставляється з іншим; обмін відбувається тоді, коли два елементи міняються місцями. Час роботи одних алгоритмів сортування зростає експоненціально, а час роботи інших логарифмічно залежить від кількості елементів;

2) час роботи в кращому та гіршому випадках. Він має значення при аналізі виконання алгоритму, якщо одна з крайових ситуацій буде зустрічатися досить часто. Алгоритм сортування часто має непоганий середній час виконання, але в гіршому випадку він працює дуже повільно;

3) поведінка алгоритму сортування. Поведінка алгоритму сортування є природною, якщо час сортування є мінімальним для вже впорядкованого списку елементів, збільшується в міру зростання ступеня невпорядкованості списку і є максимальним, коли елементи списку розташовані в зворотному порядку. Об’єм роботи алгоритму оцінюється кількістю зроблених порівнянь і обмінів.

Різні методи сортування масивів відрізняються за швидкодією. Існують прості методи сортування, які вимагають порядок порівнянь $n \times n$, де n – кількість елементів масиву та швидкі методи сортування, які вимагають порядок порівнянь $n \times \ln n$. Прості методи є зручними для пояснення принципів сортування, оскільки мають прості та короткі алгоритми. Ускладнені методи вимагають меншого числа операцій, але самі операції є більш складними, тому для невеликих масивів прості методи є більш ефективними.

Прості методи сортування можна розділити на три основні категорії:

– сортування методом «бульбашки» (простий обмін);

– сортування методом простого вибору (простий перебір);

– сортування методом простого включення (зсув-вставляння, вставляннями, вставляння та зсув).

5.3. Сортування методом «бульбашки» (простого обміну)

Найвідоміший алгоритм – бульбашкове сортування (bubble sort, сортування методом «бульбашки» або просто сортування бульбашкою). Його популярність пояснюється цікавою назвою та простотою самого алгоритму.

Алгоритм попарного порівняння елементів масиву в літературі часто називають «методом бульбашки», проводячи аналогію з бульбашкою, що піднімається з дна келиха з газованою водою. Під час спливання бульбашка стикається з іншими бульбашками і, зливаючись з ними, збільшується в об'ємі. Щоб аналогія стала очевидною, потрібно вважати, що елементи масиву розташовані вертикально один над одним, і їх потрібно так впорядкувати, щоб вони збільшувалися зверху вниз.

Алгоритм полягає у повторюваних проходах по сортованому масиву. За кожен прохід елементи послідовно порівнюються попарно і, якщо порядок у парі невірний, виконується обмін елементів. Проходи по масиву повторюються доки, поки на черговому проході не виявиться, що обміни більше не потрібні. Це буде означати, що масив вже відсортований. При проході алгоритму, елемент, що стоїть не на своєму місці, «спливає» до потрібної позиції (рис. 5.1).

У бульбашкового сортування кількість порівнянь завжди одна й та ж сама, оскільки два цикли **for** повторюються вказану кількість разів незалежно від того, був список спочатку впорядкований чи ні. Це означає, що алгоритм бульбашкового сортування завжди виконує $(n^2 - n) / 2$ порівнянь, де n – кількість елементів, для яких проводиться сортування.

Дана формула отримана на тій підставі, що зовнішній цикл виконується $n - 1$ разів, а внутрішній виконується в середньому $n / 2$ разів.

Текст програми для сортування методом «бульбашки» наведено нижче.

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#define  ARRAYSIZE    10
const int MAX_LIMIT = 80;
```

```
int mas[ARRAYSIZE];

int main(void)
{
    int buf;
    _Bool exchange;

    SetConsoleOutputCP(1251);

    srand(time(NULL));

    // ініціалізація масиву випадковими значеннями
    // з визначеного діапазону
    printf("Генеруємо масив випадкових чисел\n");
    printf("========================================\n");
    for(int i = 0; i < ARRAYSIZE; i++)
    {
        mas[i] = rand() % MAX_LIMIT + 1;
        printf("%4d", mas[i]);
    }
    printf("\n========================================\n\n");

    for(int i = ARRAYSIZE - 1; i > 0; i--)
    {
        exchange = false;
        for(int j = 0; j < i; j++)
        {
            if(mas[j] > mas[j+1])
            {
                buf      = mas[j];
                mas[j]   = mas[j+1];
                mas[j+1] = buf;
                exchange = true;
            }
        }
        if(exchange == false)
            break;
        else
        {
            for(int i = 0; i < ARRAYSIZE; i++)
                printf("%4d", mas[i]);
            printf("\n");
        }
    }
    printf("\nРезультат сортування\n");
    printf("========================================\n");
    for(int i = 0; i < ARRAYSIZE; i++)
        printf("%4d", mas[i]);
    printf("\n========================================\n");
    return 0;
}
```

Рисунок 5.1 – Сортування методом «бульбашки»

Результат роботи бульбашкового сортування наведено на рис. 5.2.

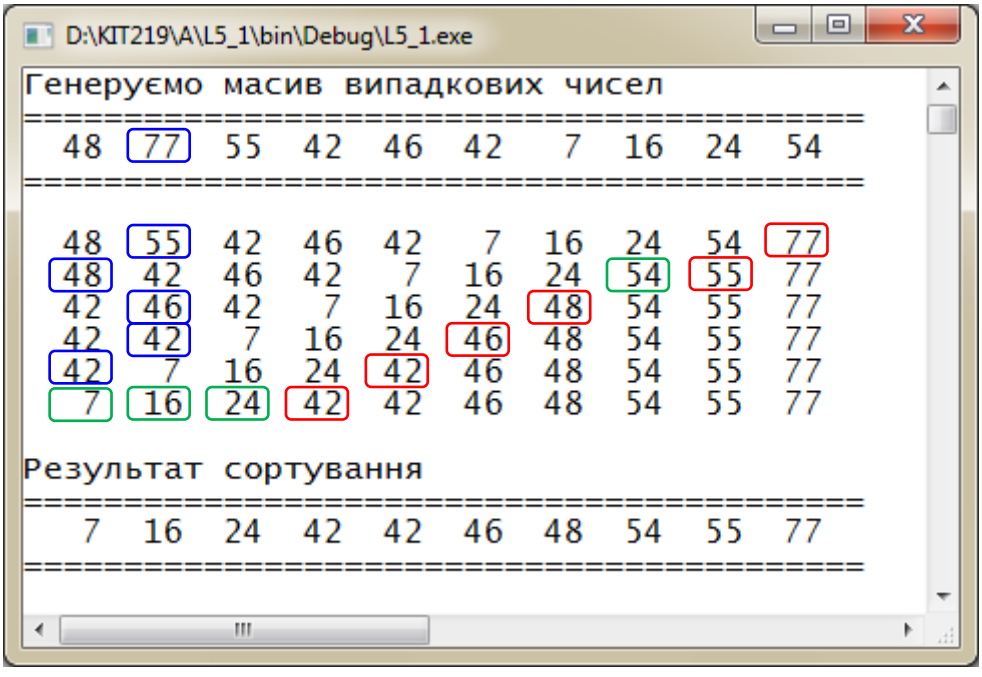


Рисунок 5.2 – Результат роботи програми сортування методом «бульбашки»

Як видно з рис. 5.2, для сортування методом «бульбашки» в даному випадку знадобилося 6 проходів, але сам принцип цього метода дана програма не дозволяє зрозуміти. Щоб зробити алгоритм бульбашкового сортування більш зрозумілим, необхідно внести до програми певні зміни. Ці зміни дозволять визначити не тільки кількість проходів, а ще й кількість замін. До того ж, ці заміни дозолять зрозуміти як саме спрацьовує алгоритм бульбашкового сортування.

Текст модифікованої програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#define ARRAYSIZE 10
const int MAX_LIMIT = 80;
int mas[ARRAYSIZE];

int main(void)
{
    int buf;
    _Bool exchange;
    int z = 0;            // кількість замін
    int p = 0;            // кількість проходів

    SetConsoleOutputCP(1251);

    srand(time(NULL));
    // ініціалізація масиву випадковими значеннями
    // з визначеного діапазону
    printf("Генеруємо масив випадкових чисел\n");
    printf("========================================\n");
    for(int i = 0; i < ARRAYSIZE; i++)
    {
        mas[i] = rand() % MAX_LIMIT + 1;
        printf("%4d", mas[i]);
    }
    printf("\n========================================\n\n");
    for(int i = ARRAYSIZE - 1; i > 0; i--)
    {
        exchange = false;
        for(int j = 0; j < i; j++)
        {
            if(mas[j] > mas[j+1])
            {
                buf      = mas[j];
                mas[j]   = mas[j+1];
                mas[j+1] = buf;
```

```
                exchange = true;
                z++;      // додаємо кількість замін
                for(int i = 0; i < ARRAYSIZE; i++)
                    printf("%4d", mas[i]);
                printf("---- j = %d\n", j);
            }
        }
        if(exchange == false)
            break;
        else
        {
            for(int i = 0; i < ARRAYSIZE; i++)
                printf("%4d", mas[i]);
            printf("------------ i = %d\n", i);
            p++;    // додаємо кількість проходів
        }
    }
    printf("\nРезультат сортування\n");
    printf("=========================================\n");
    for(int i = 0; i < ARRAYSIZE; i++)
        printf("%4d", mas[i]);
    printf("\n=========================================\n");
    printf("\nКількість замін:    %3d\n", z);
    printf("Кількість проходів: %3d\n\n", p);
    return 0;
}
```

Результат роботи модифікованої програми бульбашкового сортування наведено на рис. 5.3.

Бульбашкове сортування має таку особливість: невпорядковані елементи на «великому» кінці масиву займають правильні положення за один прохід, але невпорядковані елементи на початку масиву піднімаються на свої місця дуже повільно. Тому, замість того щоб постійно переглядати масив в одному напрямку, в послідовних проходах можна чергувати напрямки. Таким чином, елементи, які сильно віддалені від своїх положень, швидко стануть на свої місця. Дана версія бульбашкового сортування носить назву шейкер-сортування (shaker sort, сортування перемішуванням, сортування збовтуванням, сортування струшуванням), оскільки дії, які цей тип сортування проводить з масивом, нагадують збовтування або струшування. Нижче показана реалізація шейкер-сортування. Для того щоб порівняти дві версії бульбашкового сортування, ініціалізуємо масив **mas** тими самими даними, що і в попередній програмі.

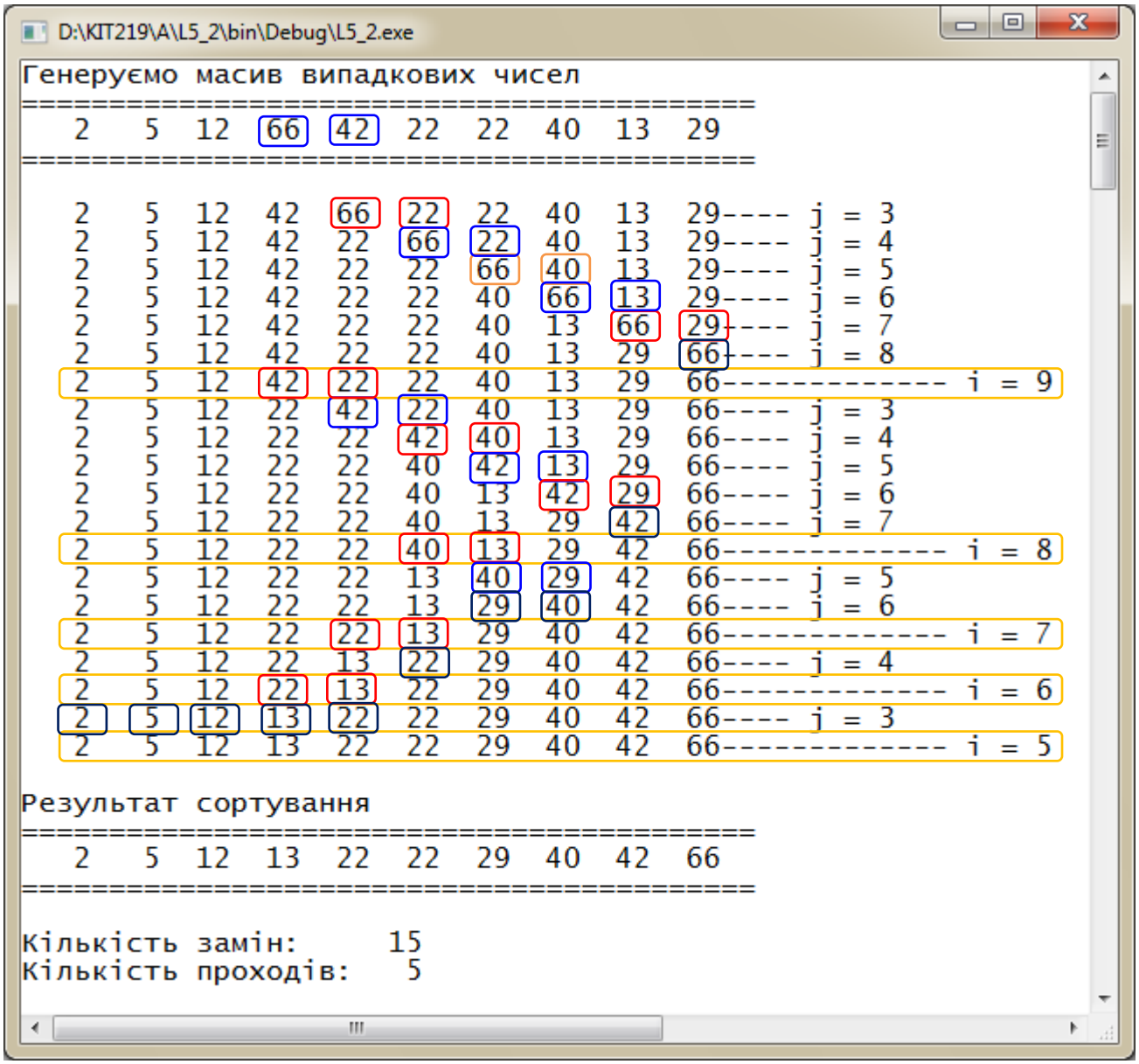


Рисунок 5.3 – Результат роботи модифікованої програми

Текст програми для шейкер-сортування має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#define ARRAYSIZE 10
const int MAX_LIMIT = 80;
int mas[ARRAYSIZE] = { 2, 5, 12, 66, 42, 22, 22, 40, 13, 29 };
int main(void)
{
    int buf;
    _Bool exchange;
```

```
int z1 = 0;     // кількість замін (зворотний прохід)
int p1 = 0;     // кількість проходів (зворотний прохід)
int z2 = 0;     // кількість замін (прямий прохід)
int p2 = 0;     // кількість проходів (прямий прохід)
SetConsoleOutputCP(1251);
printf("Вхідний масив\n");
printf("=========================================\n");
for(int i = 0; i < ARRAYSIZE; i++)
    printf("%4d", mas[i]);
printf("\n=========================================\n\n");
do
{
    exchange = false;
    for(int i = ARRAYSIZE - 1; i > 0; i--)
    {
        if(mas[i-1] > mas[i])
        {
            buf      = mas[i-1];
            mas[i-1] = mas[i];
            mas[i]   = buf;
            exchange = true;
            z1++;
            for(int i = 0; i < ARRAYSIZE; i++)
                printf("%4d", mas[i]);
            printf("---- z1 = %d\n", z1);
        }
    }
    if(exchange == false) break;
    else
    {
        for(int i = 0; i < ARRAYSIZE; i++)
            printf("%4d", mas[i]);
        printf("------------ p1 = %d\n", p1);
        p1++;
    }
    exchange = false;
    for(int i = 1; i < ARRAYSIZE; i++)
    {
        if(mas[i-1] > mas[i])
        {
            buf      = mas[i-1];
            mas[i-1] = mas[i];
            mas[i]   = buf;
            exchange = true;
            z2++;
            for(int i = 0; i < ARRAYSIZE; i++)
                printf("%4d", mas[i]);
            printf("---- z2 = %d\n", z2);
        }
    }
    if(exchange == false)
        break;
```

```
            else
            {
                for(int i = 0; i < ARRAYSIZE; i++)
                    printf("%4d", mas[i]);
                printf("------------ p2 = %d\n", p2);
                p2++;
            }
        } while(1); // сортувати доки не залишиться обмінів
        printf("\nРезультат сортування\n");
        printf("========================================\n");
        for(int i = 0; i < ARRAYSIZE; i++)
            printf("%4d", mas[i]);
        printf("\n========================================\n");
        printf("\nКількість зворотних замін:    %3d\n", z1);
        printf("Кількість зворотних проходів: %3d\n", p1);
        printf("Кількість прямих замін:       %3d\n", z2);
        printf("Кількість прямих проходів:    %3d\n", p2);
        printf("\nЗагальна кількість замін:     %3d\n", z1 + z2);
        printf("Загальна кількість проходів:  %3d\n\n", p1 + p2);
        return 0;
    }
```

Результат роботи програми для шейкер-сортування наведено на рис. 5.4.

Проаналізувавши результати роботи програм бульбашкового сортування та шейкер-сортування на однаковому вхідному наборі, можна зробити висновок, що програма шейкер-сортування зробила на один прохід менше (4 проти 5), а кількість замін (обмінів) дорівнює 15. Для отримання більш повної картини необхідно провести декілька експериментів на різних наборах.

Хоча шейкер-сортування і є покращеним варіантом у порівнянні з бульбашковим сортуванням, воно як і раніше має час виконання порядку n^2. Це пояснюється тим, що кількість порівнянь не змінилося, а кількість обмінів зменшилася лише на відносно невелику величину.

5.4. Сортування методом простого вибору (простий перебір)

Це найбільш природний алгоритм сортування. При даному методі сортування з масиву вибирається елемент з найменшим значенням і обмінюється з першим елементом. Потім з решти $n - 1$ елементів знову вибирається елемент з найменшим ключем, обмінюється з другим елементом і т. д. (рис. 5.5).

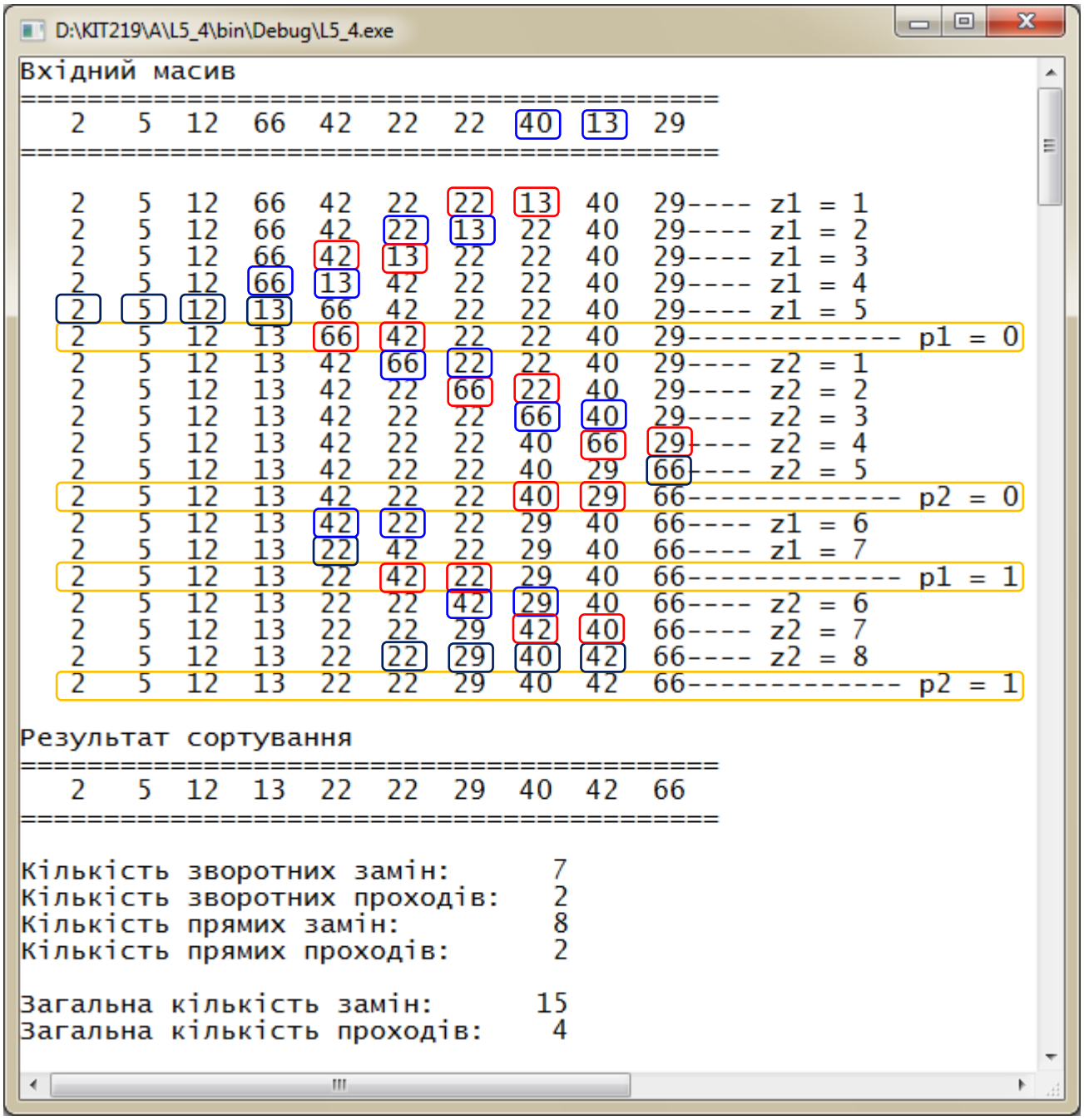


Рисунок 5.4 – Результат роботи програми для шейкер-сортування

Кроки алгоритму:

– знаходимо мінімальне значення в поточній частині масиву;

– проводимо обмін цього значення зі значенням на першій невідсортованій позиції;

– далі сортуємо хвіст масиву, виключивши з розгляду вже відсортовані елементи.

Рисунок 5.5 – Сортування методом простого вибору

Текст програми для сортування методом простого вибору має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#define ARRAYSIZE 10

const int MAX_LIMIT = 80;
int mas[ARRAYSIZE];

int main(void)
{
    _Bool exchange;
    int min, temp;
    int z = 0;            // кількість замін

    SetConsoleOutputCP(1251);
```

```
    srand(time(NULL));

    // ініціалізація масиву випадковими значеннями
    // з визначеного діапазону
    printf("Генеруємо масив випадкових чисел\n");
    printf("========================================\n");
    for(int i = 0; i < ARRAYSIZE; i++)
    {
        mas[i] = rand() % MAX_LIMIT + 1;
        printf("%4d", mas[i]);
    }
    printf("\n========================================\n\n");
    for(int i = 0; i < ARRAYSIZE - 1; i++)
    {
        // встановлюємо початкове значення мінімального
        // індексу
        min = i;
        // знаходимо мінімальний індекс елемента
        exchange = false;
        for(int j = i + 1; j < ARRAYSIZE; j++)
        {
            if(mas[j] < mas[min])
            {
                min = j;   // міняємо значення місцями
                exchange = true;
            }
        }
        if(exchange == true)
        {
            temp     = mas[i];
            mas[i]   = mas[min];
            mas[min] = temp;
            z++;
            for(int k = 0; k < ARRAYSIZE; k++)
                printf("%4d", mas[k]);
            printf("\n");
        }
    }
    printf("\nРезультат сортування\n");
    printf("========================================\n");
    for(int i = 0; i < ARRAYSIZE; i++)
        printf("%4d", mas[i]);
    printf("\n========================================\n");
    printf("\nКількість замін:    %3d\n", z);
    return 0;
}
```

Результат роботи програми для сортування методом простого вибору наведено на рис. 5.6.

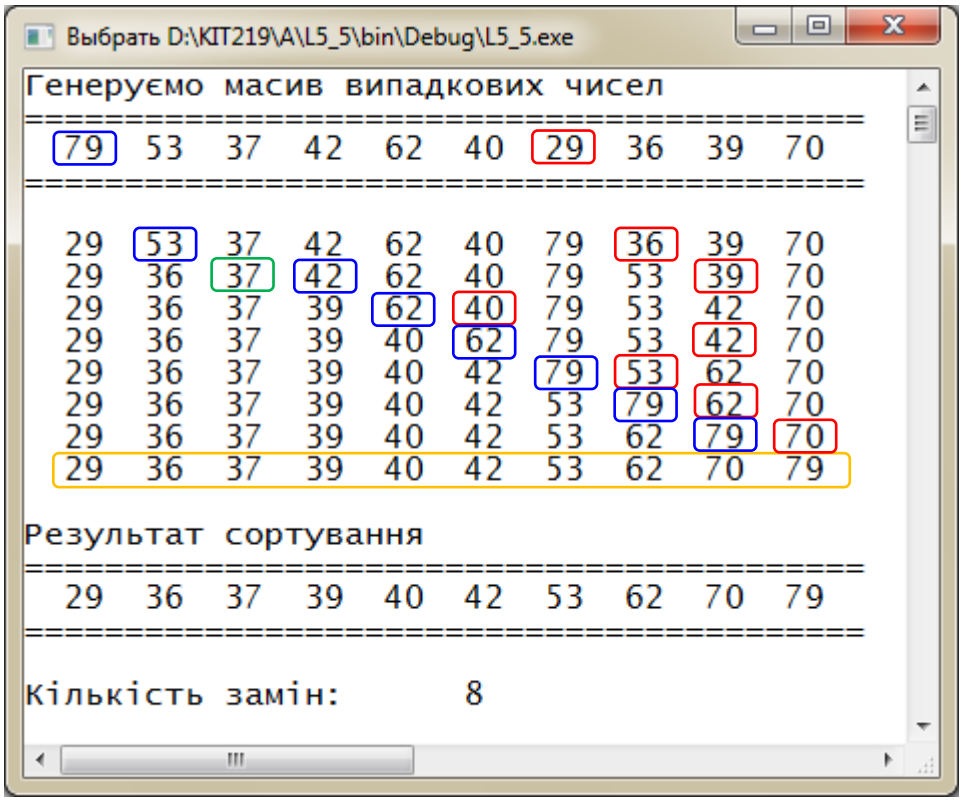


Рисунок 5.6 – Результат роботи програми для сортування методом простого вибору

Як і у бульбашкового сортування, зовнішній цикл виконується $n - 1$ разів, а внутрішній – в середньому $n / 2$ разів. Отже, сортування методом простого вибору вимагає $(n^2 - n) / 2$ порівнянь. Таким чином, це алгоритм порядку n^2, через що він вважається занадто повільним для сортування великої кількості елементів. Незважаючи на те, що кількість порівнянь у бульбашкового сортування та сортування простим вибором є однаковою, в останнього кількість обмінів у середньому випадку є набагато меншою, ніж у бульбашкового сортування.

5.5. Сортування методом простого включення (зсув-вставляння, вставляннями, вставляння і зсув)

Хоча цей метод сортування є набагато менш ефективним, ніж складні алгоритми (такі як швидке сортування), у нього є ряд переваг. Він:

– простий в реалізації;

– ефективний на невеликих наборах даних (на наборах даних до десятків елементів може виявитися кращим);

– ефективний на наборах даних, які вже частково відсортовані;

– це стійкий алгоритм сортування (не змінює порядок елементів, які вже відсортовані);

– може сортувати масив у міру його отримання;

– не вимагає тимчасової пам’яті, навіть під стек.

На кожному кроці алгоритму вибираємо один з елементів вхідних даних і вставляємо його у потрібну позицію в уже відсортованій послідовності доти, доки набір вхідних даних не буде вичерпаний. Метод вибору чергового елемен-ту з початкового масиву є довільним, тому може використовуватися практично будь-який алгоритм вибору (рис. 5.7).

Рисунок 5.7 – Сортування методом простого включення

Текст програми для сортування методом простого включення має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>
#define ARRAYSIZE 10
const int MAX_LIMIT = 80;
int mas[ARRAYSIZE];

int main(void)
{
    _Bool exchange;
    int temp;
    int z = 0;           // кількість зсувів
    int p = 0;           // кількість проходів

    SetConsoleOutputCP(1251);
    srand(time(NULL));
    //  ініціалізація масиву випадковими значеннями
    //  з визначеного діапазону
    printf("Генеруємо масив випадкових чисел\n");
    printf("========================================\n");
    for(int i = 0; i < ARRAYSIZE; i++)
    {
        mas[i] = rand() % MAX_LIMIT + 1;
        printf("%4d", mas[i]);
    }
    printf("\n========================================\n\n");
    for(int i = 0; i < ARRAYSIZE; i++) // цикл проходів
    {
        int j;

        z = 0;
        exchange = false;
        temp     = mas[i];
        // пошук місця елемента
        // робимо зсув елемента вправо, поки не дійшли
        for(j = i - 1; j >= 0 && mas[j] > temp; j--)
        {
            z++;
            mas[j+1] = mas[j];
            exchange = true;
            if(z == 1)
            {
                for(int k = 0; k < 44; k++)
                    printf(" ");
                printf("temp = %2d [%d]\n", temp, i);
            }
        }
```

```
            // місце знайдено, вставляємо елемент
            if(exchange == true)
            {
                p++;
                mas[j+1] = temp;
                for(int k = 0; k < ARRAYSIZE; k++)
                    printf("%4d", mas[k]);
                printf("  ================>%4d прохід\n", p);
            }
        }
        printf("\nРезультат сортування\n");
        printf("========================================\n");
        for(int i = 0; i < ARRAYSIZE; i++)
            printf("%4d", mas[i]);
        printf("\n========================================\n");
        printf("\nКількість проходів:    %3d\n", p);
        return 0;
    }
```

Результат роботи програми для сортування методом простого включення наведено на рис. 5.8.

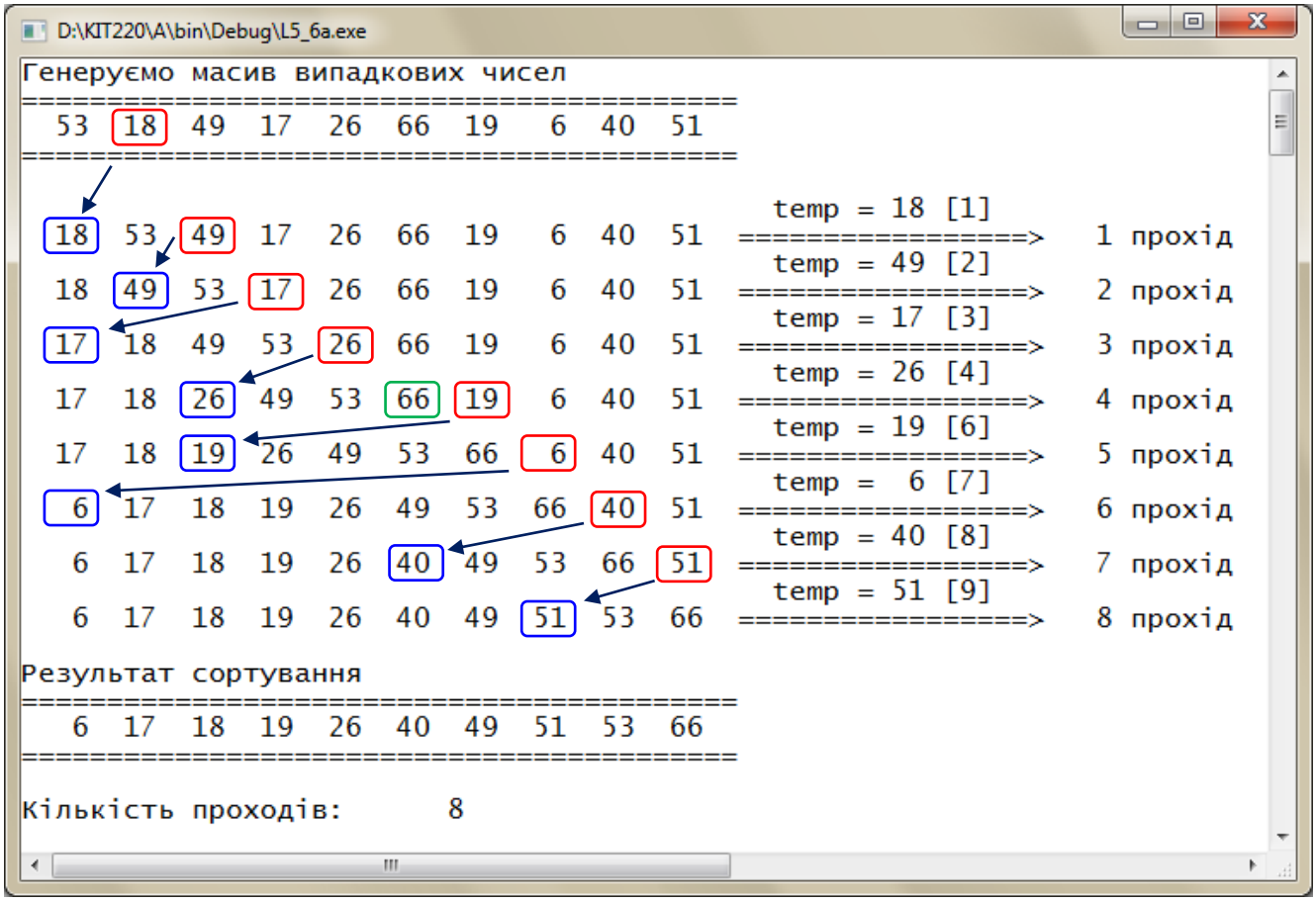


Рисунок 5.8 – Результат роботи програми для сортування методом простого включення

На відміну від бульбашкового сортування та сортування методом простого вибору, кількість порівнянь у сортуванні методом простого включення залежить від початкової впорядкованості списку. Якщо список вже відсортований, то кількість порівнянь дорівнює $n - 1$. В іншому випадку його продуктивність є величиною порядку n^2.

Хід виконання роботи:

1. Відсортуйте за неубуванням (`mas[i] ≤ mas[i+1]`) одновимірний цілочисловий масив методом «бульбашки», заданий випадковими цілими числами на проміжку [–99, 99]. Виведіть на екран на зразок рис. 5.3 (з таким самим оформленням) вхідний і відсортований масиви, а також вміст масиву на кожному кроці сортування.

2. Відсортуйте за незростанням (`mas[i] ≥ mas[i+1]`) одновимірний цілочисловий масив методом простого вибору, заданий випадковими цілими числами на проміжку [0, 50]. Виведіть на екран на зразок рис. 5.6 (з таким самим оформленням) вхідний і відсортований масиви, а також вміст масиву на кожному кроці сортування.

3. Відсортуйте за зростанням (`mas[i] < mas[i+1]`) одновимірний цілочисловий масив з 10 елементів методом простого включення, заданий з клавіатури різними двозначними числами (тобто такими, які не збігаються). Виведіть на екран на зразок рис. 5.8 (з таким самим оформленням) вхідний і відсортований масиви, а також вміст масиву на кожному кроці сортування.

4. Масив розміром $2m + 1$, де m – натуральне число, заповнений випадковими цілими числами в діапазоні [20, 90]. Знайдіть у масиві медіану. Медіаною називається елемент ряду, який ділить його на дві рівні частини: в одній знаходяться елементи, які менше медіани, в іншій – більше медіани.

5. Масив розміром m, де m – натуральне число, заповнений випадковими цілими числами в діапазоні [0, 30]. Знайдіть у масиві моду. Модою називається елемент ряду, який зустрічається найчастіше. Виведіть також інформацію про

те, скільки разів мода зустрічається у масиві. Якщо таких мод декілька, виведіть їх на консоль і вкажіть скільки разів кожна з них зустрічається.

Приклади можливих екранів консолі з результатами роботи *С*-програм для завдань 4 і 5 наведені в додатку 2.

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:
   – тексти п’яти *С*–програм;
   – результати виконання *С*-програм (5 копій екранів, перші три з яких необхідно оформити на зразок рис. 5.3 – 5.5 відповідно).
4. Висновки по роботі.

Контрольні запитання:

1. Що собою являє процес сортування?
2. Які прості методи сортування вам відомі?
3. Які існують загальні критерії оцінювання алгоритмів сортування?
4. Чим відрізняються прості методи сортування від швидких методів сортування?
5. Чому існує велика кількість алгоритмів сортування?
6. З якою метою використовуються прості сортування, якщо вони характеризуються малою ефективністю?
7. Чим відрізняється принцип сортування за неубуванням (незростанням) від сортування за зростанням (убуванням)?
8. На яких наборах вхідних даних проявляється ефективність алгоритмів простих сортувань у порівнянні між собою?
9. У чому полягає покращення методу шейкер-сортування у порівнянні з бульбашковим сортуванням?

ЛАБОРАТОРНА РОБОТА №6
ДОСЛІДЖЕННЯ СПОСОБІВ ПЕРЕДАЧІ АРГУМЕНТІВ ФУНКЦІЙ КОРИСТУВАЧА

Мета роботи: ознайомитися з методами створення та використання функцій; отримати навички роботи з функціями користувача; розглянути способи передавання параметрів і повернення результату; ознайомитися з новими поняттями («час життя» та «область видимості» змінної); отримати навички роботи з локальними та глобальними змінними.

Теоретична частина

Функцією називається послідовність інструкцій, які призначені для вирішення певної задачі. Раніше вже використовувалися бібліотечні функції вводу-виводу `printf()` і `scanf()`, а також інші функції, наприклад, математичні.

В даній лабораторній роботі будуть розглянуті правила створення власних функцій – функцій користувача.

Існує декілька причин для використання функцій користувача, по-перше, програма стає більш структурованою і, тим самим, стає більш зрозумілою та впорядкованою, по-друге, виключаються повтори схожих ділянок тексту, тобто текст програми оптимізується [4–8].

Функція може викликатися багато разів з різних частин програми. В загальному випадку вона виконує наступні дії:

- отримує параметри;
- виконує інструкції у відповідності до заданого алгоритму;
- може повертати результат до програми, яка її викликала.

6.1. Загальні поняття про функції

З використанням функцій в мові *C* пов’язані поняття, які умовно можна розділити на дві групи. До першої групи входять «визначення», «прототип» і «виклик функції» – всі три поняття пов’язані з підготовкою функції до роботи.

До другої групи належать «параметри» і «значення», які забезпечують зв'язок функції з «зовнішнім середовищем». Функція може багаторазово викликатися з різних частин програми, при цьому необхідно забезпечити її зв'язок з програмою, яка її викликала. З функції, яка викликає, до функції, яку викликають, слід передати необхідні дані, а по закінченню роботи прийняти результат.

Визначення функції – це опис дій, які буде виконувати функція відповідно до вимог алгоритму. Саме ця частина програми буде згодом багаторазово викликатися з інших частин програми.

Прототип функції (оголошення) використовується в тому випадку, якщо виклик функції передує її визначенню або, якщо визначення та виклики функції знаходяться в різних файлах.

Виклик функції забезпечує зв'язок з програмою, яка її викликала. Під час виклику:

– передаються параметри з функції, яка викликає, до функції, яку викликають;

– керування передається на першу інструкцію в тілі функції користувача;

– після завершення роботи функції до програми, що її викликала, передається значення, яке повертається (керування повертається до точки виклику).

Визначення функції складається з заголовка та тіла, наприклад:

```
double fun(int a, int b)       // заголовок функції
{
    ...                        // тіло функції
}
```

В даному прикладі визначена функція `fun()` з двома параметрами цілого типу **`int`** `a` і **`int`** `b`, яка повертає значення типу **`double`.**

Тип функції (в нашому випадку **`double`**) визначає тип значення, яке повертає функція. Якщо тип не вказано, то передбачається, що функція повертає ціле значення (типу **`int`**). Якщо функція не повинна повертати значення, то використовується тип **`void`**, який в даному випадку означає відсутність значення.

Параметри, які містяться в заголовку функції називаються формальними (в нашому випадку **`int`** `a` та **`int`** `b`). Вони служать для зв'язку функції з про-

грамою, що її викликала. Формальні параметри створюються на початку роботи функції – це локальні змінні, які ініціалізуються при виконанні функції значеннями, отриманими з програми, що її викликала. При виконанні функції параметри отримують конкретні значення. Вони називаються фактичними параметрами. Наприклад, виклик функції може виглядати наступним чином:

```
...
double z;
int s1 = 10;
...
z = fun(s1, 5); // виклик функції fun(), де
                // s1 і 5 – фактичні параметри
```

6.2. Передавання параметрів до функції та повернення значення

Параметри дозволяють передати інформацію до функції, яка викликається, з функції, яка її викликає. У тілі функції параметрами можна користуватися так само, як і локальними змінними. При виконанні функції:

– для кожного формального параметра створюється локальна змінна;

– початковими значеннями створених змінних є фактичні параметри, які визначаються при виконанні функції.

В мові *C* функція може повертати тільки одне значення, для цього її виконання слід завершити оператором **`return`**, що містить деякий вираз. Слід зазначити, що тип функції у визначенні повинен відповідати типу виразу оператора **`return`** в її тілі.

Приклад №1. Повернення функцією `sum()` результату.

```
...
int sum(int a, int b)
{
    return a + b;
}

int main(void)
{
    int x = 5, y = 10, z;

    z = sum(x, y);
    ...
}
```

При виконанні функції створюються дві локальні змінні **a** і **b**, які ініціалізуються фактичними параметрами **x** і **y**:

```
a = x
b = y
```

Функція повертає значення типу **int**, яке записується до змінної **z**.

6.3. Передавання параметрів за значенням

В усіх попередніх прикладах функції, що викликали, передавали не самі змінні, а лише їх значення. При такому виклику функція не може змінити саму змінну в програмі, яка викликається. Вона може змінити лише її тимчасову копію (параметр), значення якої втрачається при виході з функції.

Приклад №2. Функція `power()` підносить **base** до *n*-ої степені.

```
#include <stdio.h>
#define DEC 25

int power(int base, int n)
{
    int r;

    for(r = 1; n > 0; n--)
        r = r * base;
    return r;
}

int main(void)
{
    int i, p = 0;

    for(i = 0; i < 5 ; i++)
    {
        p = p + power(DEC, i);
        printf("i = %d; p = %6d\n", i, p);
    }
    return 0;
}
```

При виконанні функції **base** приймає значення `DEC`, а **n** приймає значення **i**.

Що б не відбувалося з **n** всередині функції `power()`, це не матиме жодного впливу на змінну **i** в програмі, яка викликала функцію `power()`.

Результат роботи програми наведено на рис. 6.1.

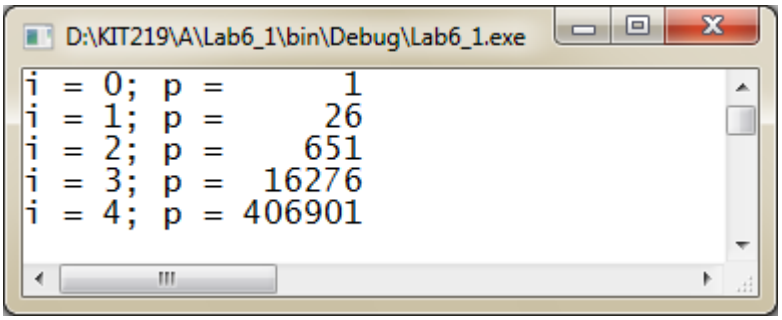


Рисунок 6.1 – Приклад передавання параметрів за значенням для функції power()

6.4. Передавання масиву до функції в якості параметру

Для того щоб організувати обробку масиву у функції користувача, необхідно повідомити їй інформацію про ім'я та розмір цього масиву:

```
int fun(char s[], int n)
{
    ...
}
```

Функція `fun()` дозволяє обробляти масиви довільного розміру. При цьому перший параметр – це ім'я масиву, а другий – його розмір.

Виклик функції буде виглядати наступним чином:

```
// Приклад обробки масиву, який містить 100 елементів
int mas[100];
fun(mas, 100);
```

6.5. Використання прототипу функції

В мові *C* визначення функцій можуть розміщуватися після визначення функції `main()`, перед ним, або знаходиться в іншому файлі.

Однак в усіх випадках (до моменту виклику) функція повинна бути визначена або оголошена. Ця вимога обумовлена тим, що компілятор повинен здійснити перевірку коректності виклику функції (перевірку відповідності кількості та типів фактичних параметрів – кількості та типам формальних параметрів). Коли виклик функції передує її визначенню, ця перевірка виконується за прототипом.

Прототип нагадує заголовок у визначенні функції:

– тіло функції відсутнє, проте наприкінці ставиться крапка з комою;

– зазначаються імена формальних параметрів, які можуть бути опущені, але типи параметрів опускати не можна.

Приклад визначення прототипу:

```
int power(int base, int n);
```

або

```
int power(int, int);
```

Якщо функція визначена до функції `main()` – прототип задавати необов'язково.

Приклад №3. Використання прототипу функції у програмі.

```
#include <stdio.h>
#include <windows.h>

int fun_max(int, int); // прототип функції

int main(void)
{
    int x, y, r;

    SetConsoleOutputCP(1251);

    printf("Введіть значення x: ");
    scanf("%d", &x);
    printf("Введіть значення y: ");
    scanf("%d", &y);
    printf("==============================\n");
    r = fun_max(x, y);  // виклик передує визначенню
    printf("r = fun_max(%2d, %2d) = %2d\n", x, y, r);
    printf("    fun_max(50, %2d) = %2d\n",
           x + y, fun_max(50, x + y));
    printf("==============================\n\n");
    return 0;
}

// визначення функції fun_max()
int fun_max(int a, int b)
{
    return (a > b) ? a : b;
}
```

Результат роботи програми наведено на рис. 6.2.

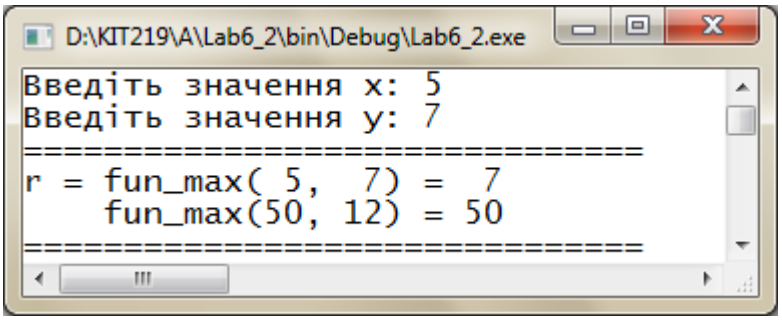


Рисунок 6.2 – Результат роботи програми

6.6. Приклади використання функцій користувача

Приклад №4. Функція, яка обчислює суму елементів масиву.

В даному випадку будемо працювати з двома різними масивами. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <time.h>
#define N  30       // розмір масиву mas
#define N1 50       // розмір масиву mas1

int sum(int array1[], int n)
{
    int res = 0;

    for(int i = 0; i < n; i++)
        res += array1[i];
    return res;
}

int main(void)
{
    // визначення масивів з 30 та 50 елементів
    int mas[N], mas1[N1];

    SetConsoleOutputCP(1251);

    srand(time(NULL));
    // ініціалізація масиву mas
    printf("Масив mas:\n\n");
    for(int i = 0; i < N; i++)
    {
        mas[i] = 2 * i + 1;
        printf("%4d", mas[i]);
        if((i+1) % 10 == 0) printf("\n");
    }
    // ініціалізація масиву mas1
    printf("\nМасив mas1:\n");
```

```
    for(int i = 0; i < N1; i++)
    {
        mas1[i] = rand() % 100;
        printf("%4d", mas1[i]);
        if((i + 1) % 10 == 0) printf("\n");
    }
    // створення змінної s і обчислення суми
    int s = sum(mas, N);
    printf("\n\nСума mas  = %d\n", s);
    printf("Сума mas1 = %d\n", sum(mas1, N1));
    return 0;
}
```

Результат роботи програми наведено на рис. 6.3.

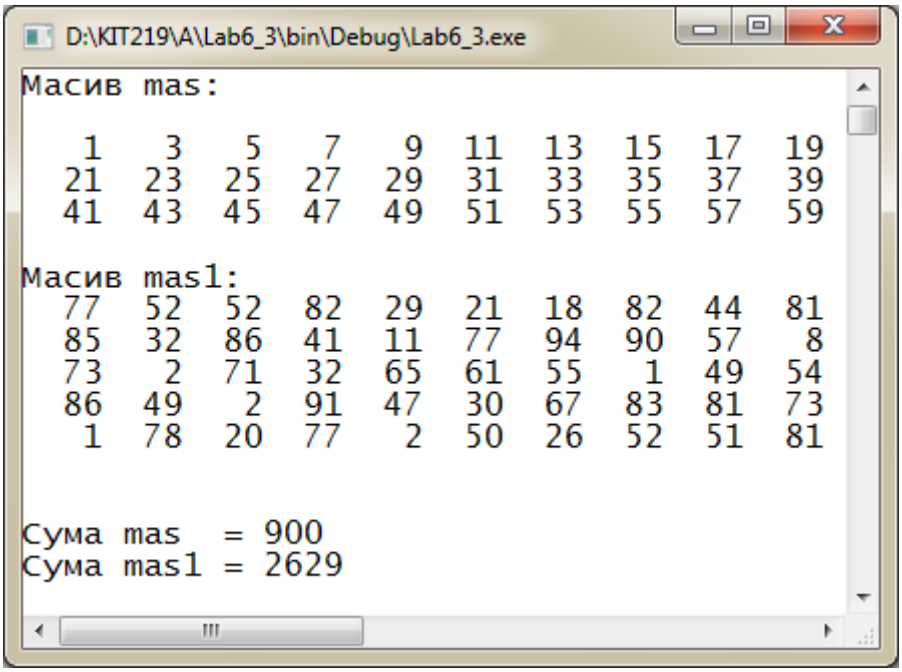


Рисунок 6.3 – Результат роботи програми, де масиви передаються в якості параметрів функції

Приклад №5. За допомогою датчика випадкових чисел заповнити матрицю дійсних чисел 5×5, вивести її на консоль, а також знайти мінімальні елементи в рядках і максимальні елементи в стовпцях.

Текст програми має такий вигляд:

```
#include <stdio.h>
#include <time.h>
#include <windows.h>
```

```
#define ROW 5             // кількість рядків
#define COL 5             // кількість стовпців

// область визначення функцій
//=========================================
// Функція для ініціалізації матриці
//=========================================
void init_matr(double mas[ROW][COL])
{
    srand(time(NULL));
    for(int i = 0; i < ROW; i++)
        for(int j = 0; j < COL; j++)
            mas[i][j] = (double)rand()/
                RAND_MAX * (4.00 - 0.51) + 0.51;
}

//=========================================
// Функція для виводу матриці на консоль
//=========================================
void print_matr(double mas[ROW][COL])
{
    for(int i = 0; i < ROW; i++)
    {
        for(int j = 0; j < COL; j++)
            printf("%7.2f ", mas[i][j]);
        printf("\n");
    }
}

//=========================================
// Функція для пошуку мінімального
// елемента в рядку
//=========================================
double row_min(double mas[ROW][COL], int n_row)
{
    double min;

    min = mas[n_row][0];
    for(int i = 0; i < COL; i++)
        if(min > mas[n_row][i])
            min = mas[n_row][i];
    return min;
}

//=========================================
// Функція для пошуку максимального
// елемента в стовпці
//=========================================
double col_max(double mas[ROW][COL], int n_col)
{
    double max;

    max = mas[0][n_col];
    for(int i = 0; i < ROW; i++)
```

```
        if(max < mas[i][n_col])
            max = mas[i][n_col];
    return max;
}

// Закінчення області визначення функцій
int main(void)
{
    // область визначення об'єктів (виділяємо пам'ять)
    double matr[ROW][COL];

    SetConsoleOutputCP(1251);

    init_matr(matr);
    printf("Матриця matr на вході:\n");
    print_matr(matr);
    printf("\nМінімальні значення в рядках:\n");
    for(int i = 0; i < ROW; i++)
        printf("Рядок №%d - %7.2f\n", i+1, row_min(matr, i));
    printf("\nМаксимальні значення в стовпцях:\n");
    for(int i = 0; i < COL; i++)
        printf("Стовпець №%d - %4.2f\n",
               i+1, col_max(matr, i));
    return 0;
}
```

Результат роботи програми наведено на рис. 6.4.

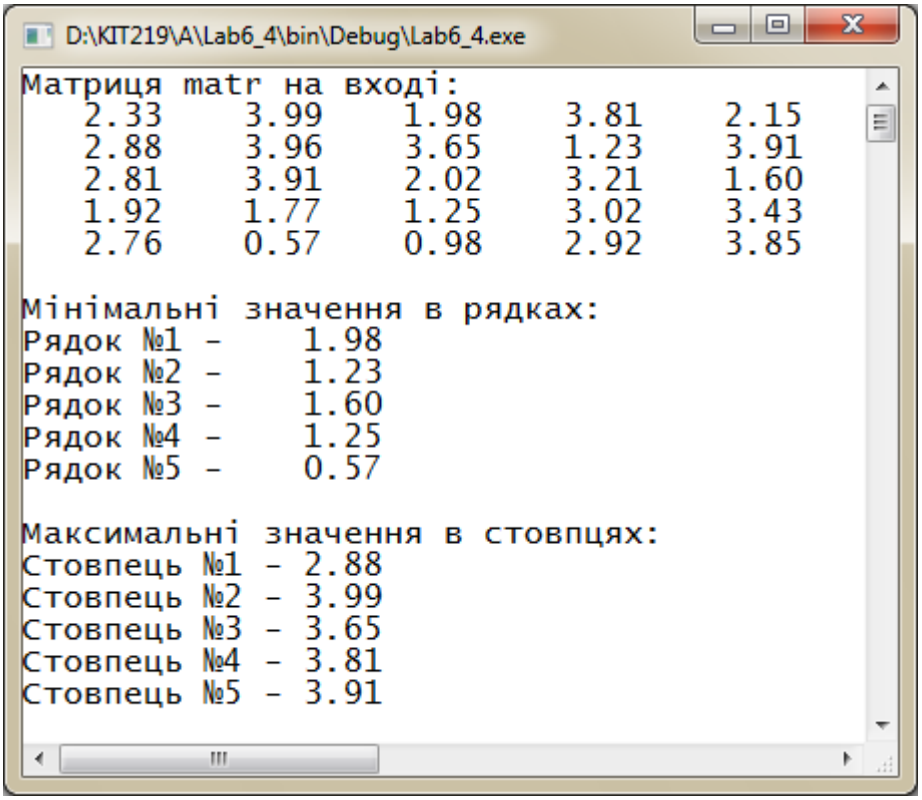


Рисунок 6.4 – Результат застосування функцій для масивів

Функції `int_matr()` і `int_matr()` не повертають значень і мають по одному параметру, який визначає матрицю. Функції `row_min()` і `col_max()` повертають результат і мають по два параметри: перший визначає матрицю, а другий – номер рядка або стовпця, які обробляються. Значення, що повертаються, без запам'ятовування виводяться на екран, коли виклик функцій `row_min()` і `col_max()` є параметрами в функціях `printf()`.

Недоліком даної реалізації є те, що всі функції працюють з матрицею фіксованого розміру **mas**`[ROW][COL]`. Цей підхід є прийнятним в тому випадку, якщо потрібно працювати з однією матрицею (або декількома матрицями однакового розміру).

Наступний приклад демонструє роботу з матрицею довільного розміру.

Приклад №6. Виконати попереднє завдання, використовуючи функції, що працюють з довільними матрицями. Продемонструвати роботу на двох матрицях різної розмірності.

Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <time.h>

// перша матриця
#define ROW1 6            // кількість рядків
#define COL1 5            // кількість стовпців

// друга матриця
#define ROW2 8            // кількість рядків
#define COL2 8            // кількість стовпців

// Область визначення функцій
//=================================================
// Функція для ініціалізації матриці r x c
//=================================================
void init_matr(double mas[], int r, int c)
{
    for(int i = 0; i < r; i++)
        for(int j = 0; j < c; j++)
            mas[i * c + j] = (double)rand()/
                RAND_MAX * (4.00 - 0.51) + 0.51;
}
//=================================================
// Функція для виводу на консоль матриці r x c
//=================================================
```

```
void print_matr(double mas[], int r, int c)
{
    for(int i = 0; i < r; i++)
    {
        for(int j = 0; j < c; j++)
            printf("%7.2f", mas[i * c + j]);
        printf("\n");
    }
}
//=================================================
// Функція для пошуку мінімального елемента
// в рядку матриці r x c
//=================================================
double row_min(double mas[], int r, int c, int n_row)
{
    double min;
    min = mas[n_row * c];
    for(int i = 0; i < c; i++)
        if(min > mas[n_row * c + i])
            min = mas[n_row * c + i];
    return min;
}

//=================================================
// Функція для пошуку максимального елемента
// в стовпці матриці r x c
//=================================================
double col_max(double mas[], int r, int c, int n_col)
{
    double max;
    max = mas[n_col];
    for(int i = 0; i < r; i++)
        if(max < mas[i * c + n_col])
            max = mas[i * c + n_col];
    return max;
}
// Кінець області визначення функцій

int main(void)
{
    // область визначення об'єктів (виділяється пам'ять)
    double matr1[ROW1 * COL1], matr2[ROW2 * COL2], cmax;

    SetConsoleOutputCP(1251);
    srand(time(0));
    init_matr(matr1, ROW1, COL1);
    printf("Матриця mas1(%dx%d):\n", ROW1, COL1);
    print_matr(matr1, ROW1, COL1);
    printf("\nМінімальні значення в рядках:\n");
    for(int i = 0; i < ROW1; i++)
        printf("Рядок №%d - %7.2f\n", i + 1,
            row_min(matr1, ROW1, COL1, i));
    printf("\nМаксимальні значення в стовпцях:\n");
```

```
    for(int i = 0; i < COL1; i++)
    {
        cmax = col_max(matr1, ROW1, COL1, i);
        printf("Стовпець №%d - %7.2f\n", i + 1, cmax);
    }
    printf("\n====================================\n\n");
    init_matr(matr2, ROW2, COL2);
    printf("Матриця mas2(%dx%d):\n", ROW2, COL2);
    print_matr(matr2, ROW2, COL2);
    printf("\nМінімальні значення в рядках:\n");
    for(int i = 0; i < ROW2; i++)
        printf("Рядок №%d - %7.2f\n", i + 1,
            row_min(matr2, ROW2, COL2, i));
    printf("\nМаксимальні значення в стовпцях:\n");
    for(int i = 0; i < COL2; i++)
    {
        cmax = col_max(matr2, ROW2, COL2, i);
        printf("Стовпець №%d - %7.2f\n", i + 1, cmax);
    }
    return 0;
}
```

Результат роботи програми наведено на рис. 6.5.

Хід виконання роботи:

1. Опрацюйте матеріал лекцій «Функції» і «Вказівники», а також теоретичний матеріал даної лабораторної роботи.

2. Напишіть *C*-програму, яка б циклічно виводила площу однієї з двох геометричних фігур (див. додаток 3). Розрахунок кожної площі фігури слід проводити за допомогою окремих функцій. Необхідно також передбачити умову виходу з нескінченого циклу натисканням визначеної вами клавіші. Завдання для вашого варіанта знаходяться в табл. 6.1. Спочатку необхідно обрати один з двох варіантів, потім для нього ввести потрібні дані та передати їх в якості параметрів до відповідної функції обчислення площі.

3. Напишіть *C*-програму для роботи з масивами цілих чисел розмірністю N×N. Значення N слід вводити з клавіатури (N = 5, 7, 9). Окремі задачі необхідно оформити у вигляді функцій. Список задач для кожного варіанта наведено в додатку 4. Приклад програми для роботи з функцією, де масив передається в якості параметру, наведено в додатку 5.

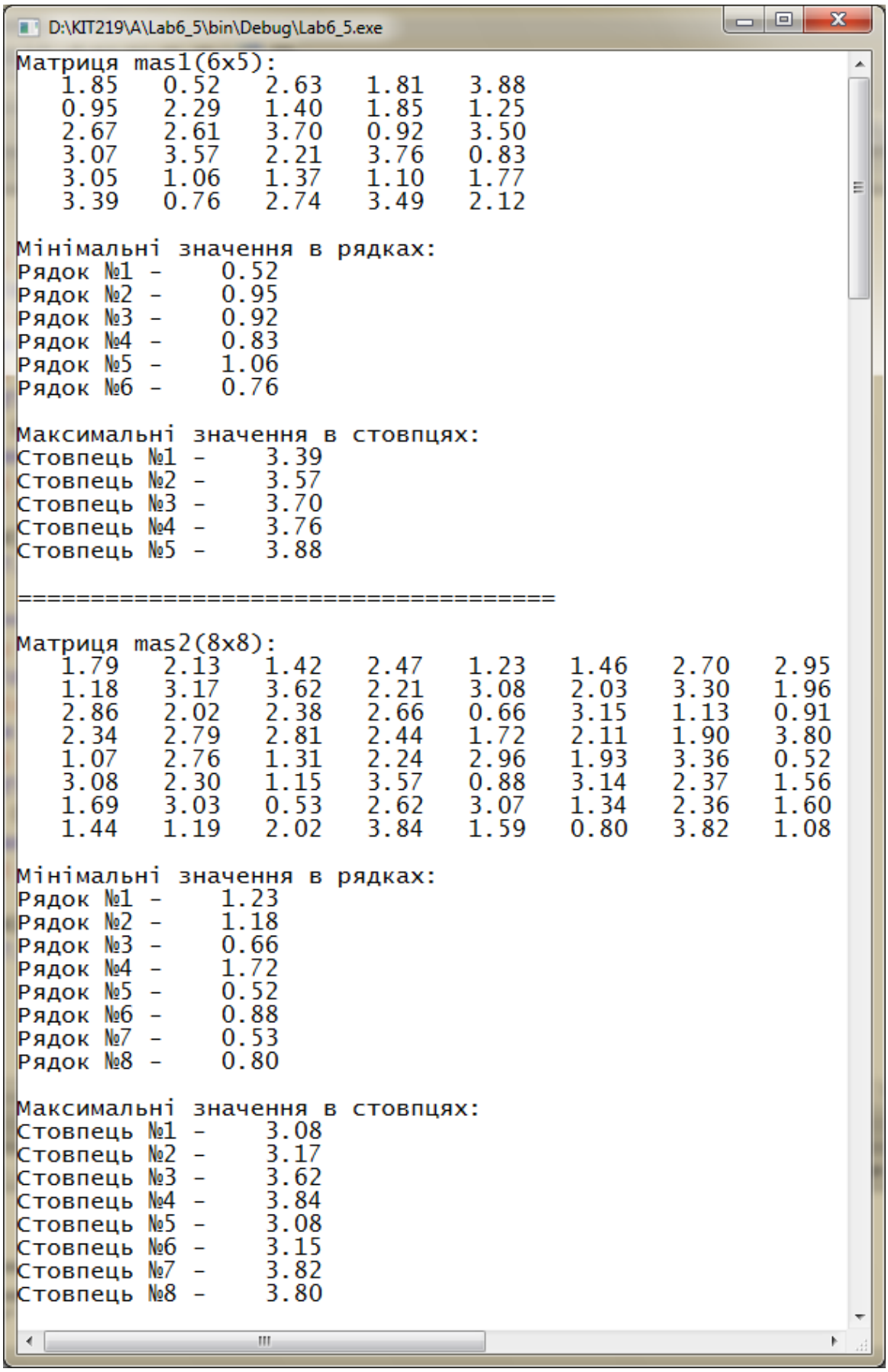


Рисунок 6.5 – Результат застосування функцій для масивів різної розмірності

Таблиця 6.1 – Вхідні дані для завдання 2 (див. додаток 3)

№ варіанта	Номери формул	№ варіанта	Номери формул	№ варіанта	Номери формул	№ варіанта	Номери формул
1	16, 17	6	11, 22	11	6, 27	16	1, 32
2	15, 18	7	10, 23	12	5, 28	17	5, 18
3	14, 19	8	9, 24	13	4, 29	18	11, 22
4	13, 20	9	8, 25	14	3, 30	19	7, 20
5	12, 21	10	7, 26	15	2, 31	20	13, 18

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:
   – тексти *C*-програм;
   – результати виконання *C*-програм.
4. Висновки по роботі.

Контрольні запитання:

1. Яким чином записується визначення функції?
2. Що таке прототип функції, коли він використовується.
3. Що таке тип функції? Які типи можна задавати для функції?
4. Чим виклик функції відрізняється від прототипу?
5. Що повертає функція, яка має тип **`void`**?
6. Яку роль виконують параметри функції?
7. Чим відрізняються формальні та фактичні параметри функції?
8. Розкажіть про використання змінних у функціях. Яка різниця між локальною та глобальною змінними?
9. Як передати масив до функції?
10. Для чого використовується ключове слово **`const`** при роботі з функціями?

ЛАБОРАТОРНА РОБОТА №7
ДОСЛІДЖЕННЯ ПРИНЦИПІВ ЗАСТОСУВАННЯ ВКАЗІВНИКІВ

Мета роботи: ознайомитися з основними принципами застосування вказівників; навчитися виконувати арифметичні операції з вказівниками та порівнювати їх між собою.

Теоретична частина

Вказівник – це змінна, яка містить адресу об’єкта. Він не несе інформації про вміст об’єкту, а лише містить відомості про те, де розміщений об’єкт.

Вказівники широко використовуються в програмуванні на мові *C*, наприклад, при роботі з масивами [4–8].

Пам’ять комп’ютера можна представити у вигляді послідовності пронумерованих однобайтових комірок, з якими можна працювати окремо або блоками [10].

Кожна змінна в пам’яті має свою адресу – номер першої комірки, де вона розташована, а також своє значення. Вказівник – це теж змінна, яка розміщується в пам’яті. Вона теж має адресу, а її значення є адресою деякої іншої змінної. Змінна, яка оголошена як вказівник, займає 4 байти в оперативній пам’яті (у разі 32-бітної версії компілятора).

Вказівник, як і будь-яка змінна, повинен бути оголошений.

Загальна форма оголошення вказівника:

```
тип_вказівника *ім’я;
```

`Тип вказівника` – це тип змінної, адресу якої він містить.

Для роботи з вказівниками на *C* визначені дві операції:

– операція `'*'` (зірочка), яка дозволяє отримати значення об’єкта за його адресою;

– операція `'&'` (амперсанд) – дозволяє визначити адресу змінної.

Наприклад:

```
char c;          // змінна
char *p;         // вказівник
p = &c;          // p = адреса c
```

Ситуацію з вказівником можна пояснити за допомогою рис. 7.1 і табл. 7.1.

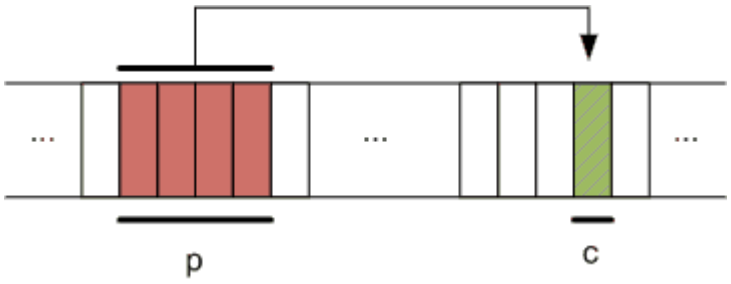


Рисунок 7.1 – Розміщення в пам'яті вказівника *p*, що посилається на змінну *c*

Таблиця 7.1 – Звернення до одних і тих самих значень змінної і адреси

Назва	Змінна	Вказівник
Адреса	`&c`	`p`
Значення	`c`	`*p`

Вказівник не є самостійним типом. Він завжди пов'язаний з якимось іншим типом. Вказівники діляться на дві категорії:

– вказівники на об'єкти;

– вказівники на функції.

Ці дві категорії вказівників відрізняються одна від одної властивостями та правилами маніпулювання. Кожен вказівник має відповідний тип.

Вказівник може бути константою або змінною, а також вказувати на константу або змінну.

Наприклад:

```
int i;                  // ціла змінна
const int ci = 1;       // ціла константа

int *pi;                // вказівник на цілу змінну
const int *pci;         // вказівник на цілу константу

int *const cpi;         // вказівник-константа на цілу змінну
const int *const cpc; // вказівник-константа на цілу константу
```

При оголошенні вказівника його можна проініціалізувати:

```
int *pi = &i;                    // вказівник на цілу змінну
const int *pci = &ci;            // вказівник на цілу константу
int *const cpi = &i;             // вказівник-константа на цілу
                                 // змінну
const int *const cpc = &ci;      // вказівник-константа на цілу
                                 // константу
```

Якщо модифікатор **const** відноситься до вказівника (тобто знаходиться між ім'ям вказівника і '*'), то він забороняє зміну значення вказівника, а якщо він знаходиться зліва від типу (тобто зліва від '*'), то він забороняє зміну значення, на яке вказує вказівник.

Способи ініціалізації вказівника

Присвоєння вказівнику адреси області пам'яті існуючого об'єкта:

– за допомогою операції одержання адреси:

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

– за допомогою проініціалізованого вказівника:

```
int *r = p;
```

Присвоєння вказівнику адреси області пам'яті в явному вигляді

```
char *cp = (char *) 0xB8000000;
```

де 0xB8000000 – шістнадцяткова константа, (**char** *) – операція приведення типу.

Присвоєння вказівнику пустого значення

```
int *N = NULL;      або      int *N = 0;
```

Специфікатор вказівника при виводі за допомогою функції printf()

Якщо на екран необхідно вивести адресу, для функції printf() слід застосовувати специфікатор %p.

Цей специфікатор формату змушує функцію printf() виводити на екран адресу, формат якої сумісний з типом адресації, яка прийнята в комп'ютері.

Операції з вказівниками

З вказівниками можна виконувати наступні операції:

– розіменування (*) - отримання значення величини, адреса якої зберігається у вказівнику;

– взяття адреси (&);

– присвоювання;

– арифметичні операції:

1) додавання вказівника тільки з константою;
2) віднімання (допускається різниця вказівників і різниця вказівника і константи);
3) інкремент (++) – збільшує значення вказівника на величину **sizeof**(тип);
4) декремент (--) – зменшує значення вказівника на величину **sizeof**(тип);
5) порівняння;
6) приведення типів.

Вказівник типу **void** вказує на область пам'яті будь-якого розміру. Розіменування такого вказівника необхідно проводити з операцією приведення типів.

Приклад №1. Демонстрація ситуації, коли вказівники різних типів вказують на одне й те ж саме місце в пам'яті, однак при розіменуванні виходять різні результати. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>

int main(void)
{
    unsigned long L = 12345678;
    char *cp        = (char *)&L;
    int  *ip        = (int *)&L;
    long *lp        = (long *)&L;

    SetConsoleOutputCP(1251);

    printf("Значення адреси комірки пам\'яті, "
           "в якій знаходиться змінна L\n");
    printf("&L  = %10p\n\n", &L);
    printf("Значення змінної L\n");
    printf("L   = %10ld\n\n", L);
    printf("Значення змінної, на яку вказує "
           "вказівник cp типу char\n");
```

```
    printf("*cp = %10c\n\n", *cp);
    printf("Значення змінної, на яку вказує "
           "вказівник ip типу int\n");
    printf("*ip = %10d\n\n", *ip);
    printf("Значення змінної, на яку вказує "
           "вказівник lp типу long\n");
    printf("*lp = %10ld\n\n", *lp);
    return 0;
}
```

Результат виконання програми наведено на рис. 7.2.

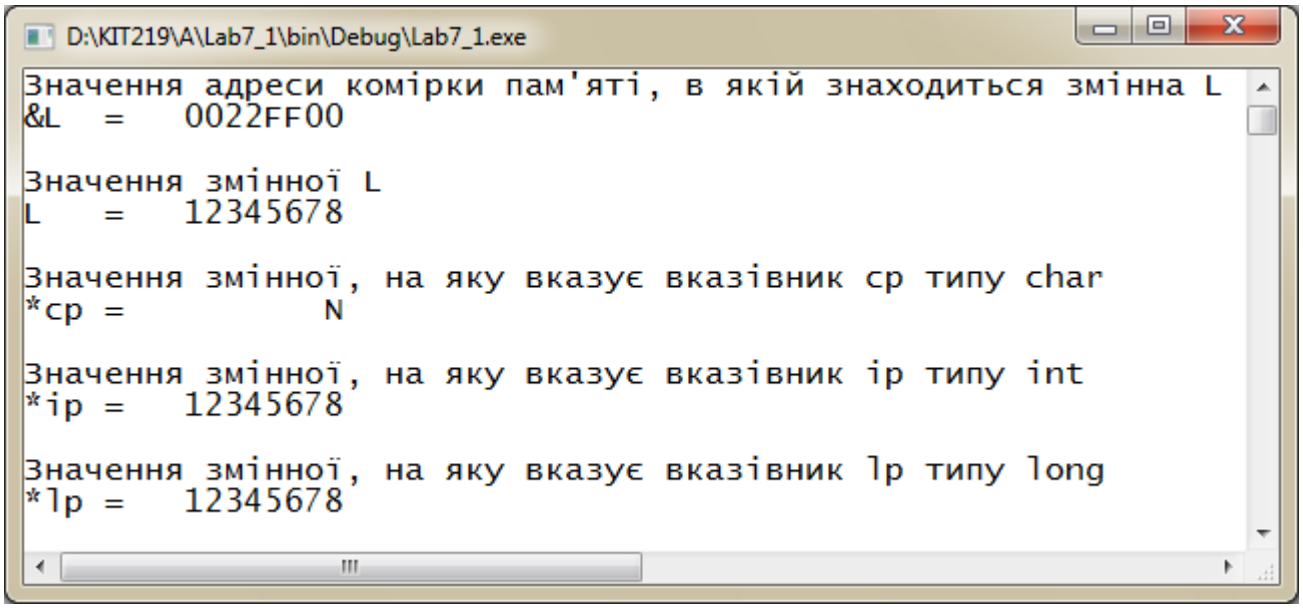


Рисунок 7.2 – Демонстрація роботи програми з вказівниками різного типу

Приклад №2. Виконання операцій з вказівниками.

```
#include <stdio.h>
#include <windows.h>

int main(void)
{
     int a, c, b;
     int *ca, *cb;
     int *sa, *sb, *sc;

     SetConsoleOutputCP(1251);

     printf("a = ");
     scanf("%d", &a);
     printf("b = ");
     scanf("%d", &b);
     c = a + b;
```

```
    printf("c = a + b = %2d\n", c);
    // ініціалізація вказівників через взяття адреси
    sa = &a;
    sb = &b;
    sc = &c;
    // присвоювання вказівнику значення виразу
    *sc = c + a - b;
    // присвоювання вказівнику значення іншого вказівника
    ca = sa;
    cb = sb;
    // присвоювання вказівнику значення константи
    *sa = 12;
    printf("\n*ca = %2d\n", *ca);
    printf("*sa = %2d\n", *sa);
    printf("*cb = %2d\n", *cb);
    printf("*sb = %2d\n", *sb);
    printf("*sc = %2d\n\n", *sc);
    printf("ca = %p\n", ca);
    printf("cb = %p\n", cb);
    printf("sc = %p\n\n", sc);
    printf("a = %2d\n", a);
    printf("b = %2d\n", b);
    printf("c = %2d\n\n", c);
    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("&c = %p\n\n", &c);

    printf("*&a = %2d\n\n", *&a);

    printf("*cb - *ca = %2d\n", *cb - *ca);
    printf("*cb + *ca = %2d\n\n", *cb + *ca);

    printf("Значення вказівника до\n"
           "додавання константи 2: cb = %p\n\n", cb);
    cb = cb + 2;   // додавання константи до вказівника
    printf("Значення вказівника після\n"
           "додавання константи 2: cb = %p\n\n", cb);
    cb++;
    printf("cb = %p\n", cb);
    ca--;
    printf("ca = %p\n", ca);
    printf("cb - ca = %d\n\n", cb - ca);
    return 0;
}
```

Результат виконання програми наведено на рис. 7.3.

Вказівники одного й того ж самого типу можна порівнювати за допомогою стандартних операцій порівняння. При цьому порівнюються значення вказівників, а не значення величин, на які вони посилаються.

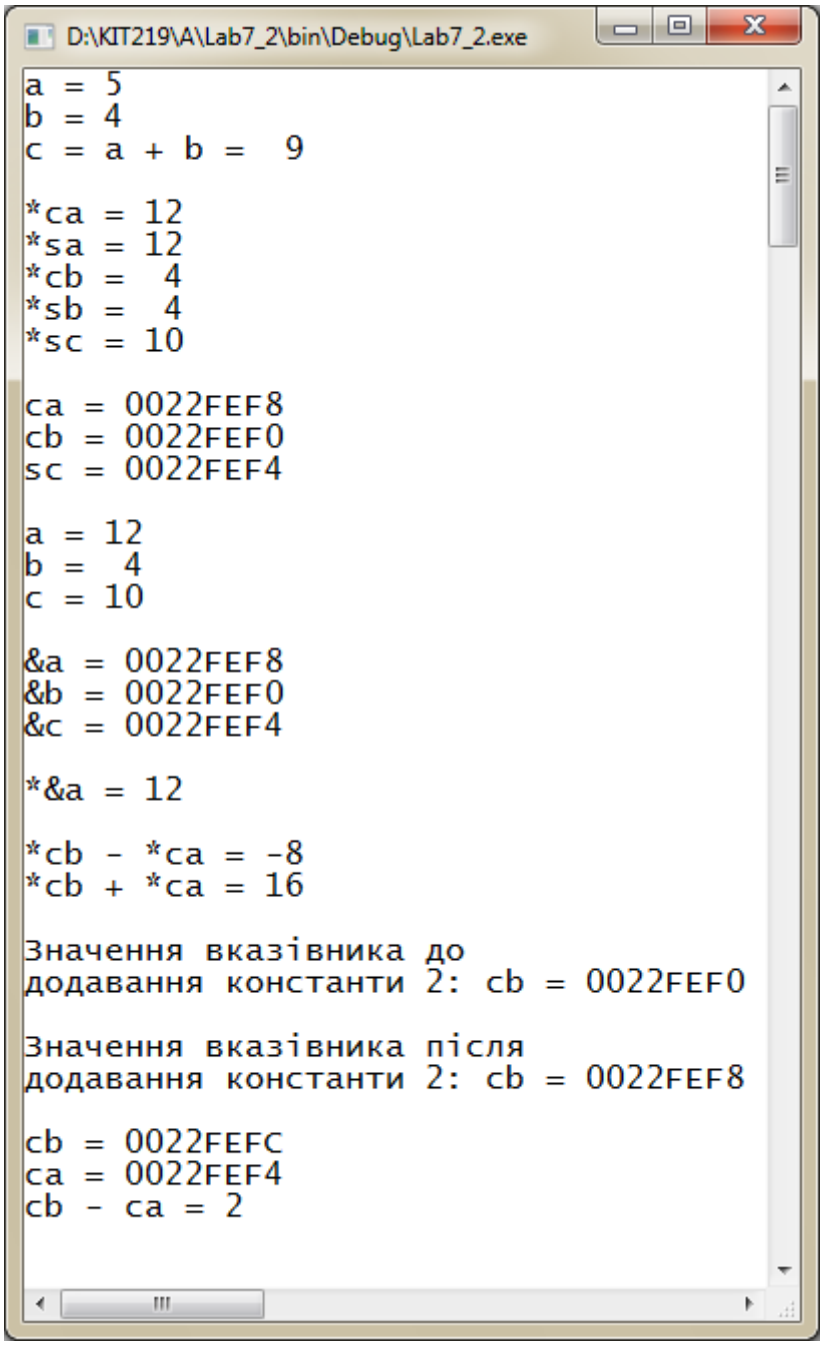


Рисунок 7.3 – Виконання арифметичних операцій з вказівниками

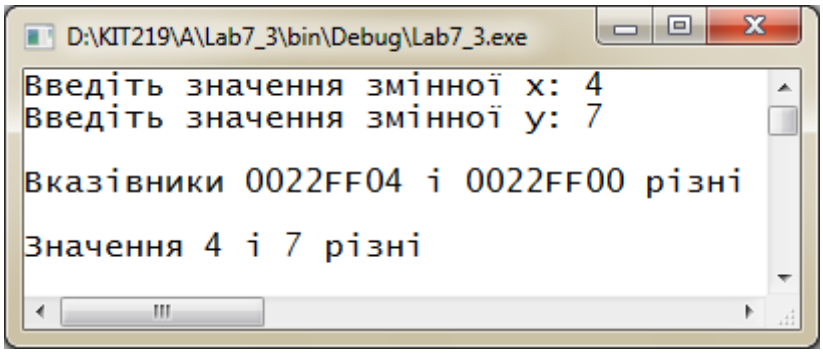
Приклад №3. Порівняння вказівників.

Текст програми для порівняння вказівників має такий вигляд:

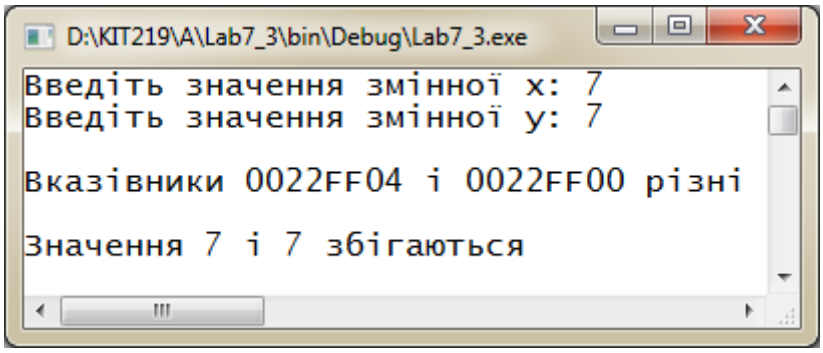
```
#include <stdio.h>
#include <windows.h>

int main(void)
{
    int x, y;
    int *ptrx = &x;
    int *ptry = &y;
```

```
    SetConsoleOutputCP(1251);
    printf("Введіть значення змінної x: ");
    scanf("%d", &x);
    printf("Введіть значення змінної y: ");
    scanf("%d", &y);
    printf("\n");
    // Порівнюємо вказівники
    if(ptrx == ptry)
        printf("Вказівники %p і %p збігаються\n", ptrx, ptry);
    else
        printf("Вказівники %p і %p різні\n", ptrx, ptry);
    printf("\n");
    // Порівнюємо значення, на яке вказують вказівники
    if(*ptrx == *ptry)
        printf("Значення %d і %d збігаються\n", x, y);
    else
        printf("Значення %d і %d різні\n", x, y);
    printf("\n");
    return 0;
}
```

Результати роботи програми наведені на рис. 7.4.



а)



б)

Рисунок 7.4 – Порівняння вказівників

Хід виконання роботи:

1. Опрацюйте матеріал лекції «Вказівники», а також теоретичний матеріал даної лабораторної роботи.

2. Для програми з прикладу №1 введіть значення `L` відповідно до вашого варіанта (табл. 7.2). Поясніть у висновках кожен рядок результатів, що отримані на екрані.

3. У програмі визначте та ініціалізуйте змінну типу **`double`**, вказівник на тип **`double`** і вказівник на тип **`void`**. Виконайте присвоєння вказівникам адреси змінної. Виведіть на консоль адресу змінної, значення вказівників і значення, які отримані після розіменування вказівників. Для демонстрації ролі та послідовності виконання унарних операцій отримання адреси `'&'` і розіменування `'*'`, виведіть на консоль значення виразу **`"* &`** `ім'я_змінної`**`"`**.

4. Для програми з прикладу №2 введіть числа **`a`** і **`b`** відповідно до вашого варіанта (табл. 7.2). Поясніть у висновках кожен рядок отриманих результатів.

5. Напишіть *C*-програму, яка передбачає ввід з клавіатури натурального чотиризначного числа та обчислює в циклі суму всіх його цифр, використовуючи вказівники та операції над ними.

6. Визначте та ініціалізуйте в *C*-програмі змінну типу **`double`**. Визначте вказівники **`char`** `*`, **`int`** `*`, **`double`** `*`, **`void`** `*` та ініціалізуйте їх адресою змінної. Виведіть на консоль значення вказівників, їх розміри та довжини ділянок пам'яті, які пов'язані з виразами розіменування вказівників.

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:
   – тексти *C*-програм;
   – результати виконання *C*-програм.
4. Висновки по роботі.

Контрольні запитання:

1. Чому вказівник не може існувати як самостійний тип?

2. З якою метою в програмі може бути використаний вказівник типу **`void`**?

3. Що буде результатом розіменування вказівника типу **`void`** без приведення типів?

4. Як зміниться значення вказівника після застосування до нього операції інкременту (декременту)?

5. Чому для вказівників визначені додавання та віднімання тільки з цілими константами?

6. У чому відмінність вказівника на константу від вказівника-константи?

7. Два вказівника різних типів вказують на одне й те ж саме місце в пам’яті. Порівняйте результати операцій розіменування та взяття адреси з такими вказівниками. Порівняйте значення вказівників.

8. Якщо об’єкт займає в пам’яті кілька байтів, то яку адресу має значення вказівника на цей об’єкт?

Таблиця 7.2 – Вхідні дані для написання програм

№ варіанта	*L*	*a*	*b*	№ варіанта	*L*	*a*	*b*
1	73645387	8	2	11	18234219	3	5
2	82750731	5	1	12	18272131	1	9
3	61162859	7	9	13	20098736	2	7
4	75839211	3	8	14	28711632	6	4
5	28374654	2	6	15	18371209	5	8
6	11706354	4	7	16	19087132	8	9
7	11116271	3	6	17	24565387	7	3
8	31182072	3	5	18	42751131	4	2
9	77625921	2	9	19	61160059	2	5
10	25332876	9	4	20	18894217	5	9

ЛАБОРАТОРНА РОБОТА №8
ДОСЛІДЖЕННЯ ОСНОВНИХ МОЖЛИВОСТЕЙ РОБОТИ З ФАЙЛАМИ

Мета роботи: ознайомитися з основними принципами роботи з файлами; навчитися застосовувати функції для роботи з файлами: `fopen()`, `fclose()`, `fread()`, `fwrite()`та інші; вивчити режими роботи з файлами для функції `fopen()`.

Теоретична частина

8.1. Робота з файлами

Стандартна бібліотека *C* містить набір функцій для роботи з файлами. Ці функції описані в стандарті ANSI. Файловий ввід-вивід не є частиною мови *C*, а ANSI-функції – це не єдиний засіб вводу-виводу [4–8]. Функції ANSI-бібліотеки підтримуються усіма *C*-компіляторами, і тому програми, що їх застосовують, легко можна перенести з однієї платформи на іншу. Прототипи функцій вводу-виводу та типи даних, які використовуються для цього, описані в стандартному заголовному файлі **`stdio.h`**.

8.2. Відкриття файлу

Для доступу до файлу застосовується тип даних **`FILE`**. Це структурний тип, ім'я якого задано за допомогою оператора **`typedef`** в стандартному заголовному файлі **`stdio.h`**. Програмісту не треба знати, як побудована структура типу файл: вона може залежати від системи, тому з метою перенесення програм звертатися явно до полів структури **`FILE`** заборонено. Тип даних «вказівник на структуру **`FILE`**» застосовується в програмах як чорний ящик: функція відкриття файлу повертає цей вказівник у випадку успіху. У подальшому усі файлові функції застосовують цей вказівник для доступу до файлу.

Прототип функції відкриття файлу виглядає наступним чином:

```
FILE *fopen(const char *path, const char *mode);
```

У визначенні цього прототипу **path** – це шлях до файлу (наприклад, ім'я файлу або абсолютний шлях до файлу), **mode** – режим відкриття файлу.

Рядок **mode** може містить декілька букв:

– "r" (від слова read) – означає, що файл відкривається для читання (файл повинен існувати);

– "w" (від слова write) – означає запис до файлу, при цьому старий вміст файлу втрачається, а у випадку відсутності файлу він створюється;

– "a" (від слова append) – означає запис в кінець існуючого файлу або створення нового файлу з подальшим записом, якщо файл не існує.

В деяких операційних системах є розбіжності в роботі з текстовими та бінарними файлами (до таких систем відносяться MS DOS і MS Windows). В них при відкритті бінарного файлу до рядка **mode** слід додати букву "b" (від слова binary), а при відкритті текстового файлу – букву "t" (від слова text). Крім цього, при відкритті можна дозволити виконувати як операції читання, так і запису. Для цього використовується символ "+" (плюс). Порядок букв в рядку **mode** такий: спочатку йде одна з букв "r", "w", "a", потім у довільному порядку можуть йти символи "b", "t", "+". Букви "b" і "t" можна використовувати, навіть якщо в операційній системі немає розбіжностей між бінарними та текстовими файлами, в даному випадку вони просто ігноруються.

Значення символів в рядку **mode** наведені в табл. 8.1.

Таблиця 8.1 – Режими роботи з файлами для функції `fopen()`

Режим	Дія
"r"	Відкрити існуючий файл для читання.
"w"	Відкрити файл для запису. Старий вміст файлу втрачається. У разі відсутності файлу він створюється.
"a"	Відкрити файл для запису. Якщо файл існує, то запис відбувається в його кінець.
"t"	Відкрити текстовий файл.
"b"	Відкрити бінарний (двійковий) файл.
"+"	Дозволити і читання, і запис.

Розглянемо декілька прикладів відкриття файлів:

```
FILE *f, *g, *h;
. . .
// 1 Відкрити текстовий файл "input.txt" для читання
f = fopen("input.txt", "rt");

// 2 Відкрити бінарний файл "c:\Windows\Temp\tmp.dat"
// для читання та запису
g = fopen("c:/Windows/Temp/tmp.dat", "wb+");

// 3 Відкрити текстовий файл "c:\Windows\Temp\input.log"
// для дописування в кінець файлу
h = fopen("c:\\Windows\\Temp\\input.log", "at");
```

Зверніть увагу, що в рядкових константах *C*, які містять шлях до файлу, зворотну скісну риску треба повторювати двічі, як це зроблено в третьому прикладі. Хоча, стандартна бібліотека *C* дозволяє в іменах файлів використовувати нормальну скісну риску замість зворотної (ця можливість була використана у другому прикладі).

У разі успіху функція відкриття файлу `fopen()` повертає ненульовий вказівник на структуру типу **`FILE`**, що описує параметри відкритого файлу. Цей вказівник треба потім використовувати в усіх файлових операціях. У випадку неуспіху (наприклад, при намаганні відкрити для читання неіснуючий файл) повертається нульовий вказівник. При цьому глобальна системна змінна **`errno`**, що описана в стандартному заголовному файлі **`errno.h`**, містить числовий код помилки. У разі неуспіху при відкритті файлу цей код можна роздрукувати, щоб отримати додаткову інформацію:

```
#include <stdio.h>
#include <errno.h>
. . .
FILE *f = fopen("filename.txt", "rt");
if(f == NULL)
{
    printf("Помилка відкриття файлу з кодом %d\n", errno);
    . . .
}
```

8.3. Константа NULL

В наведеному вище прикладі при відкритті файлу функція `fopen()` у випадку помилки повертає нульовий вказівник на структуру **`FILE`**. Щоб перевірити, чи виникла помилка чи ні, слід порівняти значення, що повертається, з с нульовим вказівником. Для наочності стандартний заголовний файл **`stdio.h`** визначає символічну константу `NULL` як нульовий вказівник на тип **`void`**:

```
#define NULL ((void *) 0)
```

Зроблено це для того, щоб відрізнити число нуль від нульового вказівника. При цьому мова *C*, в якій контроль помилок здійснюється недостатньо строго, дозволяє порівнювати вказівник загального типу **`void`** `*` з будь-яким іншим вказівником. Між тим, в *C* замість константи `NULL` завжди можна використовувати просто 0, і навряд чи від цього програма стає менш зрозумілою. Константа `NULL` не є частиною мови *C*, і без підключення одного зі стандартних заголовних файлів, в якому вона визначається, використовувати її неможна.

8.4. Діагностика помилок

Використовувати змінну **`errno`** для виводу коду помилки не дуже зручно, оскільки необхідно мати під рукою таблицю можливих кодів помилок та їх значень. В стандартній бібліотеці *C* існує більш зручна функція `perror()`, яка виводить системне повідомлення про останню помилку замість її коду. Вивід здійснюється англійською мовою, але є можливість додати до системного повідомлення будь-який текст, який вказується в якості єдиного аргументу функції `perror()`. Наприклад, попередній фрагмент переписується наступним чином:

```
#include <stdio.h>
. . .
FILE *f = fopen("filename.txt", "rt");
if(f == 0)
{
    perror("Не можу відкрити файл для читання");
    . . .
}
```

Функція `perror()` виводить спочатку повідомлення користувача про помилку, потім після двокрапки системне повідомлення. Наприклад, при виконанні наведеного фрагменту у випадку помилки через відсутність файлу буде виведено

```
Не можу відкрити файл для читання: No such file or directory
```

8.5. Функції бінарного читання та запису

Після того як файл відкритий, можна читати інформацію з файлу або записувати інформацію до файлу. Розглянемо спочатку функції бінарного читання `fread()` та запису `fwrite()`. Вони називаються бінарними тому, що не виконують жодного перетворення інформації при вводі або виводі (з одним невеликим виключенням при роботі з текстовими файлами, яке буде розглянуто нижче): інформація зберігається в файлі як послідовність байтів рівно в тому вигляді, в якому вона зберігається в пам'яті комп'ютера.

Функція читання `fread()` має такий прототип:

```
size_t fread(
    char *buffer,        // Масив для читання даних
    size_t elemSize,     // Розмір одного елемента
    size_t numElems,     // Число елементів для читання
    FILE *f              // Вказівник на структуру FILE
);
```

Тут **`size_t`** визначений як беззнаковий цілий тип в системних заголовних файлах. Функція намагається прочитати **`numElems`** елементів з файлу, який задається вказівником **`f`** на структуру **`FILE`**, розмір кожного елемента дорівнює **`elemSize`**. Функція повертає реальне число прочитаних елементів, яке може бути меншим, ніж **`numElems`**, у випадку кінця файлу або помилки читання. Вказівник **`f`** повинен бути повернений функцією `fopen()` в результаті успішного відкриття файлу.

Розглянемо приклад використання функції `fread()`.

```
FILE *f;
double buff[100];
size_t res;
```

```
f = fopen("tmp.dat", "rb"); // Відкриваємо файл
if(f == 0)
{   // У разі помилки під час відкриття файлу
    // вивести повідомлення про помилку
    perror("Не можу відкрити файл для читання");
    exit(1);  // Завершити роботу з кодом 1
}

// Намагаємось прочитати 100 дійсних чисел з файлу
res = fread(buff, sizeof(double), 100, f);
// res дорівнює реальній кількості прочитаних чисел
```

В цьому прикладі файл **`tmp.dat`** відкривається для читання як бінарний, з нього читається 100 дійсних чисел розміром 8 байт кожне.

Функція `fread()` повертає реальну кількість прочитаних чисел, яка менша або дорівнює 100.

Функція `fread()` читає інформацію у вигляді потоку байтів і в незмінному вигляді поміщає її в пам'ять. Слід розрізняти текстове представлення чисел та їх бінарне представлення. В наведеному вище прикладі числа в файлі повинні бути записані в бінарному вигляді, а не у вигляді тексту. Для текстового вводу чисел слід використовувати функції вводу по формату, які будуть розглянуті нижче.

Відкриття файлу як текстового за допомогою функції `fopen()`, наприклад,

```
FILE *f = fopen("tmp.dat", "rt");
```

зовсім не означає, що числа при вводі за допомогою функції `fopen()` будуть перетворюватися з текстової форми на бінарну. З цього випливає тільки те, що в операційних системах, в яких рядки текстових файлів розділяються парами символів `"\r\n"` (вони мають назви `CR` и `LF` – повернення каретки та вставити рядок, Carriage Return і Line Feed). При вводі такі пари символів замінюються на один символ `"\n"` (новий рядок). Навпаки, при виводі символ `"\n"` замінюється на пару `"\r\n"`. Такими операційними системами є MS DOS і MS Windows.

Функція зворотного запису до файлу `fwrite()` аналогічна функції читання `fread()`. Вона має такий прототип:

```
size_t fwrite(
    char *buffer,        // Масив даних, які записуються
    size_t elemSize,     // Розмір одного елемента
    size_t numElems,     // Число елементів, які записуються
    FILE *f              // Вказівник на структуру FILE
);
```

Функція повертає кількість реально записаних елементів, яка може бути меншою ніж **numElems**, якщо під час запису виникла помилка – наприклад, не вистачило вільного простору на диску.

Приклад використання функції `fwrite()`:

```
FILE *f;
double buff[100];
size_t num;
. . .
f = fopen("tmp.res", "wb"); // Відкриваємо файл "tmp.res"
if(f == 0)
{   // У разі помилки під час відкриття файлу
    // вивести повідомлення про помилку
    perror("Не можу відкрити файл для запису");
    exit(1);  // Завершити роботу програми з кодом 1
}

// Записуємо до файлу 100 дійсних чисел
num = fwrite(buff, sizeof(double), 100, f);
// У разі успіху num == 100
```

8.6. Закриття файлу

По закінченні роботи з файлом його треба обов’язково закрити. Система зазвичай забороняє повний доступ до файлу доки, поки він не буде закритий. В нормальному режимі система забороняє одночасний запис у файл для двох різних програм. Крім того, інформація реально записується повністю у файл лише в момент його закриття. До цього вона може міститися в оперативній пам’яті (в так званій файловій кеш-пам’яті), що при виконанні численних операцій запису та читання значно прискорює роботу програми.

Для закриття файлу використовується функція `fclose()` з прототипом

```
int fclose(FILE *f);
```

У випадку успіху функція `fclose()` повертає нуль, а у разі помилки – від'ємне значення (точніше, константу кінець файлу `EOF`, що визначена в системних заголовних файлах як мінус одиниця). При помилці можна скористатися функцією `perror()`, щоб вивести причину помилки. Відзначимо, що помилка під час закриття файлу – явище дуже рідке (чого не скажеш по відношенню до відкриття файлу), тому аналізувати значення, що повертається функцією `fclose()`, не обов'язково.

Приклад застосування функції `fclose()`:

```
FILE *f;
f = fopen("tmp.res", "wb"); // Відкриваємо файл "tmp.res"
if(f == 0)
{   // У разі помилки відкриття файлу
    // вивести повідомлення про помилку
    perror("Не можу відкрити файл для запису");
    exit(1);  // Завершити роботу програми з кодом 1
}
. . .
// Закриваємо файл
if(fclose(f) < 0)
{
    // Вивести повідомлення про помилку
    perror("Помилка під час закриття файлу");
}
```

В якості прикладу використання розглянутих вище функцій файлового вводу наведемо програму, яка підраховує кількість символів і рядків у заданому текстовому файлі.

Програма спочатку вводить ім'я файлу з клавіатури. Для цього використовується функція `scanf()` вводу по формату з вхідного потоку. Для вводу рядка застосовується формат %s. Потім файл відкривається для читання як бінарний (це означає, що при читанні не буде відбуватися жодного перетворення роздільників рядків). Використовуючи в циклі функцію читання `fread()`, зчитується вміст файлу порціями по 512 байтів, кожного разу збільшуючи сумарне число прочитаних символів. Після читання чергової порції сканується масив

прочитаних символів і підраховується кількість символів "\n" переходу на новий рядок, які записані наприкінці рядків текстових файлів в MS-DOS або MS Windows. Після цього файл закривається і результат виводиться на консоль.

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

int main(void)
{
    char fileName[256]; // Містить шлях до файлу
    FILE *f;            // Структура, що описує файл
    char buff[512];     // Масив для вводу символів
    size_t num;         // Число прочитаних символів
    int numChars = 0;   // Сумарна кількість символів
    int numLines = 0;   // Сумарна кількість рядків
    int i;              // Змінна циклу

    SetConsoleOutputCP(1251);

    printf("Введіть ім'я файлу: ");
    scanf("%s", fileName);

    f = fopen(fileName, "rb"); // Відкриваємо файл для читання
    if(f == 0)
    {   // У разі помилки відкриття файлу
        // вивести повідомлення про помилку
        perror("Неможливо відкрити файл для читання");
        exit(1);  // Закінчити роботу програми з кодом 1,
                  // тобто завершення з помилкою
    }

    while((num = fread(buff, 1, 512, f)) > 0)
    {
        // Читаємо блок розміром 512 символів
        // num - число реально прочитаних символів
        // Цикл продовжується, поки num > 0
        numChars += num; // Збільшуємо число символів
        // Підраховуємо кількість символів '\n'
        for(i = 0; i < num; i++)
        {
            if(buff[i] == '\n')
            {
                numLines++; // Збільшуємо кількість рядків
            }
        }
    }
    fclose(f);

    // Виводимо результат
    printf("Кількість символів у файлі = %d\n", numChars);
```

```
    printf("Кількість рядків у файлі   = %d\n", numLines);
    return 0; // Повертаємо код успішного завершення
}
```

Розглянемо приклад виконання програми, коли вона застосовується для власного тексту *C*-програми, що записаний у файлі **`main.c`**.

Результат виконання програми наведено на рис. 8.1.

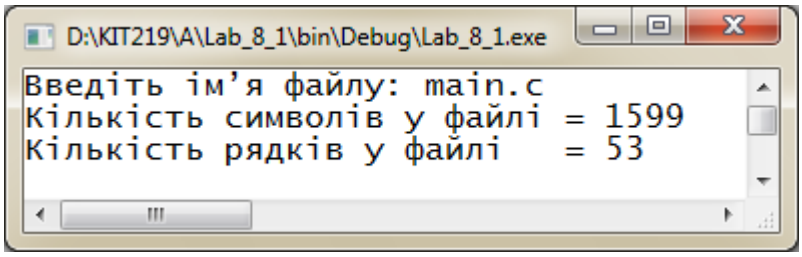


Рисунок 8.1 – Результат виконання програми підрахунку кількості символів і рядків у файлі `main.c`

8.7. Форматний файловий ввід-вивід

На відміну від функції бінарного вводу `fread()`, яка вводить байти з файлу без усякого перетворення безпосередньо в пам'ять комп'ютера, функція форматного вводу `fscanf()` призначена для вводу інформації з перетворенням її з текстового представлення на бінарне. Припустимо, що інформація записана в текстовому файлі в звичному для людини вигляді (тобто таким чином, що її можна прочитати або ввести до файлу, застосовуючи текстовий редактор). Функція `fscanf()` читає інформацію з текстового файлу та перетворює її у внутрішнє представлення даних у пам'яті комп'ютера. Інформація про кількість елементів, що читаються, їх типи та особливості представлення задається за допомогою формату. У випадку функції вводу формат – це рядок, який містить описи одного або декількох елементів, що вводяться. Формати, що використовуються функцією `fscanf()`, аналогічні до тих, які застосовувалися функцією `scanf()` і вже неодноразово розглядалися. Кожен елемент формату починається з символу процента "%". Формати, які використовуються при вводі найчастіше наведені в табл. 8.2.

Прототип функції `fscanf()` має такий вигляд:

```
int fscanf(FILE *f, const char *format, ...);
```

Три крапки тут означає, що функція має змінну кількість аргументів (більше двох), і що кількість і типи аргументів, починаючи з третього, є довільними. Насправді, фактичні аргументи, починаючи з третього, повинні бути вказівниками на змінні, що вводяться.

Таблиця 8.2 – Формати, які найчастіше використовуються при вводі

Формат	Опис
%d	Ціле десяткове число типу **int** (d – від decimal).
%lf	Дійсне число типу **double** (lf – від long float).
%c	Один символ типу **char**.
%s	Ввід рядка. З вхідного потоку виділяється слово, що обмежене пробілами або символами нового рядка `'\n'`. Слово поміщається до масиву символів. Кінець слова позначається нульовим байтом.

Розглянемо декілька прикладів використання функції `fscanf()`.

```
int n, m;
double a;
char c;
char str[256];
FILE *f;
. . .
fscanf(f, "%d", &n);        // Вводимо ціле число
fscanf(f, "%lf", &a);       // Вводимо дійсне число
fscanf(f, "%c", &c);        // Вводимо один символ
fscanf(f, "%s", str);       // Вводимо рядок (виділяється
                            // чергове слово з вхідного потоку)
fscanf(f, "%d%d", &n, &m); // Вводимо два цілих числа
```

Функція `fscanf()` повертає число успішно введених елементів. Таким чином, значення, що повертається, завжди буде меншим або дорівнювати кількості процентів всередині форматного рядка (яке дорівнює числу фактичних аргументів мінус 2).

Функція `fprintf()` використовується для форматного виводу до файлу. Дані при виводі перетворюються на їх текстове представлення відповідно до

форматного рядка. Його відмінність від форматного рядка, що використовувався у функції вводу `fscanf()`, полягає в тому, що він може містити не тільки формати для перетворення даних, але й звичайні символи, які записуються без перетворення до файлу. Формати, як і у випадку функції `fscanf()`, починаються з символу процента "%". Вони аналогічні форматам, що використовуються функцією `fscanf()`. Невеличка різниця полягає в тому, що формати функції `fprintf()` дозволяють також керувати представленням даних, наприклад, вказувати кількість позицій, що відводяться під запис числа, або кількість цифр після десяткової крапки при виводі дійсного числа. Деякі типові приклади форматів для виводу наведені в табл. 8.3.

Таблиця 8.3 – Приклади форматів для виводу

Формат	Опис
%d	Вивід цілого десяткового числа.
%10d	Вивід цілого десяткового числа. Для запису числа відводиться 10 позицій. Запис при необхідності доповнюється пробілами зліва.
%lf	Вивід дійсного числа типу **`double`** у формі з фіксованою десятковою крапкою.
%.3lf	Вивід дійсного числа типу **`double`** з трьома знаками після десяткової крапки.
%12.3lf	Вивід дійсного числа типу **`double`** з трьома знаками після десяткової крапки. Під число відводиться 12 позицій.
%c	Вивід одного символу.
%s	Вивід рядка, тобто масиву символів. Кінець рядка задається нульовим байтом.

Прототип функції `fprintf()` має такий вигляд:

```
int fprintf(FILE *f, const char *format, ...);
```

Три крапки, як і у випадку функції `fscanf()`, означають, що функція має змінну кількість аргументів. Кількість і типи аргументів, починаючи з третього, повинні відповідати форматному рядку. На відміну від функції `fscanf()`, фак-

тичні аргументи, починаючи з третього, являють собою значення, що виводяться, а не вказівники на змінні. Для прикладу розглянемо невелику програму, яка виводить дані у файл **tmp.dat**. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <math.h>
#include <string.h>

int main(void)
{
    int n = 4, m = 6;
    double x = 2.;
    char str[256] = "Тестове завдання";

    SetConsoleOutputCP(1251);

    FILE *f = fopen("tmp.dat", "wt");  // Відкрити файл
    if(f == 0)                         // для запису
    {
        perror("Неможливо відкрити файл для запису");
        return 1; // Завершити програму з кодом помилки
    }
    fprintf(f, "n = %d, m = %d\n", m, n);
    fprintf(f, "x = %.4lf, sqrt(x) = %.4lf\n", x, sqrt(x));
    fprintf(f, "Рядок \"%s\" містить %d символів.\n",
        str, strlen(str));
    fclose(f);  // Закрити файл
    return 0;   // Успішне завершення програми
}
```

В результаті виконання цієї програми до файлу **tmp.dat** буде записаний текст, який наведено на рис. 8.2.

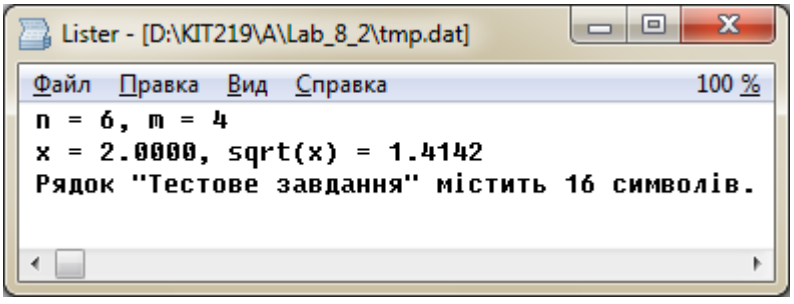


Рисунок 8.2 – Приклад застосування функції `fprintf()`

В останньому прикладі форматний рядок містить всередині подвійні лапки. Це спеціальні символи, що виконують роль обмежувачів рядка, тому всере-

дині рядка їх треба екранувати (тобто захищати від інтерпретації як спеціальних символів) за допомогою зворотної скісної риски `'\'`.

В програмі використовується стандартна функція `sqrt()`, що обчислює квадратний корінь числа, і стандартна функція `strlen()`, яка обчислює довжину рядка.

8.8. Потік вводу-виводу

В операційних системах MS DOS і MS Windows використовується поняття потоку вводу або виводу. Потік являє собою послідовність байтів. Розрізняють потоки вводу та виводу. Програма може читати дані з потоку вводу і виводити дані в потік виводу.

В *C* робота з потоком не відрізняється від роботи з файлом. Доступ до потоку відбувається за допомогою змінної типу **`FILE`** `*`. В момент початку роботи *C*-програми відкриваються три потоки:

– **`stdin`** – стандартний вхідний потік (за замовчуванням він асоціюється з клавіатурою);

– **`stdout`** – стандартний вихідний потік (за замовчуванням він асоціюється з екраном терміналу);

– **`stderr`** – вихідний потік для виводу інформації про помилки (також асоціюються з екраном терміналу).

Змінні **`stdin`**, **`stdout`**, **`stderr`** є глобальними. Вони описані в стандартному заголовному файлі **`stdio.h`**. Операції файлового вводу-виводу можуть використовувати ці потоки. Наприклад, рядок

```
fscanf(stdin, "%d", &n);
```

вводить значення цілочислової змінної **`n`** з вхідного потоку. Рядок

```
fprintf(stdout, "n = %d\n", n);
```

виводить значення змінної **`n`** у вихідний потік. Рядок

```
fprintf(stderr, "Помилка під час відкриття файлу\n");
```

виводить вказаний текст в потік **stderr**, що використовується зазвичай для виводу повідомлень про помилки. Функція `perror()` також виводить повідомлення про помилки в потік **stderr**.

За замовчуванням, стандартний вихідний потік і вихідний потік для виводу помилок призначені для виводу на екран терміналу.

Стандартний вхідний потік має номер 0, стандартний вихідний потік – номер 1, стандартний потік виводу помилок – номер 2.

Таким чином, існування двох різних потоків виводу дозволяє при необхідності спрямовувати нормальний вивід і вивід інформації про помилки в різні файли.

8.9. Функції текстового перетворення

Стандартна бібліотека вводу-виводу *C* має також дві функції вводу та виводу `sscanf()` і `sprintf()` не у файл або потік, а в рядок символів (тобто масив байтів), що розташований у пам'яті комп'ютера. Мнемоніка функцій наступна: в назві функції `fscanf()` перша буква `'f'` означає файл (file), тобто ввід відбувається з файлу. Відповідно, в назві функції `sscanf()` перша буква `'s'` означає рядок (string), тобто ввід відбувається з текстового рядка. Остання буква `'f'` в назвах цих функцій означає «форматний». Першим аргументом функцій `sscanf()` і `sprintf()` є рядок (тобто масив символів, обмежений нульовим байтом), з якого відбувається ввід або в який відбувається вивід. Цей рядок немов стоїть замість файлової змінної в функціях `fscanf()` і `fprintf()`.

Функції `sscanf()` і `sprintf()` є зручними для перетворення даних з текстового представлення у внутрішнє та навпаки. Наприклад, в результаті виконання фрагменту

```
char txt[256] = "-135.76";
double x;
sscanf(txt, "%lf", &x);
```

текстовий запис дійсного числа, що міститься в рядку **txt**, перетворюється у

внутрішнє представлення дійсного числа, результат записується у змінну **x**.

Навпаки, при виконання фрагменту

```
char txt[256];
int x = 12345;
sprintf(txt, "%d", x);
```

значення цілочислової змінної **x** буде перетворено в текстову форму і записано в рядок **txt**. В результаті рядок буде містити текст "12345", обмежений нульовим байтом.

Для перетворення даних з текстового представлення на внутрішнє в стандартній бібліотеці *C* є також функції `atoi()` і `atof()` з прототипами

```
int atoi(const char *txt);       // текст => int
double atof(const char *txt);    // текст => double
```

Функція `atoi()` перетворює текстове представлення цілого числа типу **int** на внутрішнє. Відповідно, функція `atof()` перетворює текстове представлення дійсного числа типу **double**. Мнемоніка імен наступна:

– `atoi` означає «character to integer»;

– `atof` означає «character to float».

В останньому випадку **float** слід розуміти як число з рухомою комою, тобто дійсне число, що має тип **double**, а зовсім не **float**.

Прототипи функцій `atoi()` і `atof()` описані в стандартному заголовному файлі **stdlib.h**, а не **stdio.h**, тому при використанні цих функцій треба підключати цей файл:

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

Відзначимо, що аналогів функції `sprintf()` для зворотного перетворення з внутрішнього на текстове представлення в стандартній бібліотеці *C* немає. Компілятор *C* фірми Borland має функції `itoa()` та `ftoa()`, однак, ці функції не входять до стандарту й іншими компіляторами не підтримуються, тому користуватися ними не слід.

8.10. Корисні функції вводу-виводу

Стандартна бібліотека вводу-виводу мови *C* містить ряд інших корисних функцій вводу-виводу (табл. 8.4).

Таблиця 8.4 – Функції стандартної бібліотеки вводу-виводу *C*

Функції	Призначення
Ввід-вивід символу	
`int fgetc(FILE *f);`	Ввести символ з потоку `f`
`int fputc(int c, FILE *f);`	Вивести символ у потік `f`
Ввід-вивід рядка	
`char *fgets(char *line,int size, FILE *f);`	Ввести рядок з потоку `f`
`char *fputs(char *line, FILE *f);`	Вивести рядок у потік `f`
Позиціонування у файлі	
`int fseek(FILE *f, long offset, int whence);`	Встановити поточну позицію у файлі `f`
`long ftell(FILE *f);`	Отримати поточну позицію у файлі `f`
`int feof(FILE *f);`	Перевірити, чи досягнуто кінця файлу `f`

Функція `fgetc()` повертає код введеного символу або константу `EOF` (визначену як мінус одиниця) у випадку кінця файлу або помилки читання. Функція `fputc()` записує один символ до файлу. У разі помилки `fputc()` повертає константу `EOF` (тобто від'ємне значення), у разі успіху – код символу **`c`**, що виводиться (не від'ємне значення).

Наведемо приклад використання функції `fgetc()`. Для цього перепишемо програму, яка була розглянута раніше та підраховувала кількість символів і рядків у текстовому файлі. Текст програми буде мати такий вигляд:

```
#include <stdio.h>
#include <windows.h>

int main(void)
{
    char fileName[256];     // Шлях до файлу
```

```
    FILE *f;                // Структура, що описує файл
    int c;                  // Код символу
    int numChars = 0;       // Сумарна кількість символів
    int numLines = 0;       // Сумарна кількість рядків

    SetConsoleOutputCP(1251);

    printf("Введіть ім'я файлу: ");
    scanf("%s", fileName);
    // Відкриваємо файл для читання
    f = fopen(fileName, "rb");
    if(f == 0) // Якщо виникла помилка під час відкриття
    {          // файлу, вивести повідомлення про помилку
        perror("Неможливо відкрити файл для читання");
        return 1; // Закінчити роботу програми з кодом 1
    }
    while((c = fgetc(f)) != EOF) // Читаємо символ
    {   // Цикл продовжується, поки c != -1 (кінець файлу)
        numChars++; // Збільшуємо кількість символів на 1
        // Підраховуємо кількість символів нового рядка
        if(c == '\n')
        {
            numLines++; // Збільшуємо кількість рядків
        }
    }
    fclose(f);
    printf("Кількість символів у файлі = %d\n", numChars);
    printf("Кількість рядків у файлі   = %d\n", numLines);
    return 0; // Повертаємо код успішного завершення
}
```

Результат роботи програми для демонстрації можливостей використання функції `fgetc()` наведено на рис. 8.3.

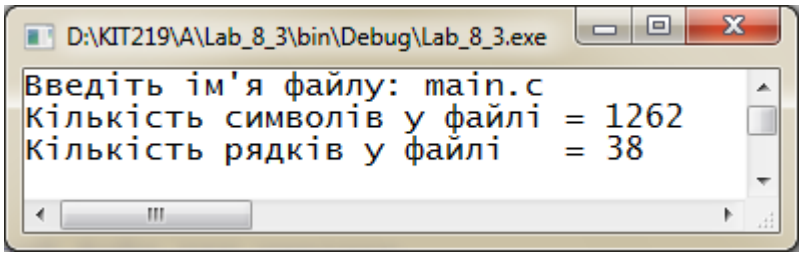


Рисунок 8.3 – Результат застосування функції `fgetc()`

Функція `fgets()` з прототипом

```
char *fgets(char *line, int size, FILE *f);
```

виділяє з файлу або вхідного потоку **f** черговий рядок і записує його до масиву

символів **line**. Другий аргумент **size** вказує розмір масиву для запису рядка. Максимальна довжина рядка на одиницю менше, ніж **size**, оскільки завжди наприкінці зчитаного рядка додається нульовий байт. Функція сканує вхідний потік доки, поки не зустріне символ нового рядка "\n" або поки число введених символів не стане дорівнювати **size–1**. Символ нового рядка "\n" також записується до масиву безпосередньо перед нульовим байтом.

Функція повертає вказівник **line** у разі успіху або нульовий вказівник у разі помилки або кінця файлу.

Раніше до стандартної бібліотеки *C* входила також функція gets() з прототипом **char** *gets(**char** *line), яка зчитувала черговий рядок зі стандартного вхідного потоку та поміщала його до масиву, адреса якого була його єдиним аргументом. Відмінність від функції fgets() полягає в тому, що не вказується розмір масиву **line**. В результаті, подавши на вхід функції gets() дуже довгий рядок, можна отримати переповнення масиву та стерти або підмінити фрагмент пам'яті, що використовується програмою. Якщо програма має привілеї суперкористувача, то застосування в ній функції gets() відкриває шлях до зламу системи, який може використовуватися хакерами. Тому на сьогоднішній день функція gets() вважається небезпечною і застосування її не рекомендовано. Замість gets() слід використовувати fgets() з третім аргументом **stdin.**

При виконанні файлових операцій система підтримує вказівник поточної позиції у файлі. Під час читання або запису **n** байтів вказівник поточної позиції збільшується на **n.** Таким чином, читання або запис відбуваються послідовно. Бібліотека вводу-виводу *C* надає можливість порушувати цю послідовність шляхом позиціонування в довільну точку файлу. Для цього використовується стандартна функція fseek() з прототипом

```
int fseek(FILE *f, long offset, int whence);
```

Перший аргумент **f** функції визначає файл, для якого відбувається операція позиціонування. Другий аргумент **offset** задає зміщення в байтах. Воно

може бути як додатним, так і від'ємним. Третій аргумент **whence** вказує, звідки відраховувати зміщення. Він може приймати одне з трьох значень, що задані як цілі константи в стандартному заголовному файлі **stdio.h** (табл. 8.5).

Таблиця 8.5 – Константи, які задають зміщення у файлі

Константа	Значення	Пояснення
SEEK_SET	0	Зміщення, яке відраховується від початку файлу.
SEEK_CUR	1	Зміщення, яке відраховується від поточної позиції.
SEEK_END	2	Зміщення, яке відраховується від кінця файлу.

Наприклад, функція

```
fseek(f, 0, SEEK_SET);
```

встановлює поточну позицію на початок файлу.

Функція

```
fseek(f, -4, SEEK_END);
```

встановлює поточну позицію на чотири байти раніше ніж кінець файлу.

Нарешті, функція

```
fseek(f, 12, SEEK_CUR);
```

пересовує поточну позицію на 12 байтів вперед.

Відзначимо, що зміщення може бути додатним навіть при використанні константи SEEK_END (тобто при позиціонуванні відносно кінця файлу): в цьому випадку при наступному записі розмір файлу відповідно збільшиться.

Функція повертає нульове значення у випадку успіху та від'ємне значення EOF (дорівнює –1) у протилежному випадку. Наприклад, якщо вказане зміщення є некоректним при виконанні заданої операції або якщо файл чи потік не дозволяє виконати пряме позиціонування.

Дізнатися про поточну позицію відносно початку файлу можна за допомогою функції ftell() з прототипом

```
long ftell(FILE *f);
```

Функція `ftell()` повертає поточну позицію (не від'ємне значення) у випадку успіху або від'ємне значення –1 у разі неуспіху (наприклад, якщо файл не дозволяє пряме позиціонування).

Нарешті, дізнатися, чи знаходиться поточна позиція в кінці файлу, можна за допомогою функції `feof()` з прототипом

```
int feof(FILE *f);
```

Вона повертає ненульове значення (тобто істину), якщо кінець файлу досягнутий, і нульове значення (тобто хибність) в іншому випадку. Наприклад, в наступному фрагменті в циклі перевіряється, чи досягнутий кінець файлу, і, якщо ні, зчитується черговий байт:

```
FILE *f;
. . .
while(!feof(f))
{     // цикл поки не кінець файлу
    int c = fgetc(f); // прочитати черговий байт
    . . .
}                     // кінець циклу
```

Хід виконання роботи:

1. Опрацюйте матеріал лекцій «Функції для роботи з файлами» та «Двійковий файловий ввід-вивід», а також теоретичний матеріал даної лабораторної роботи.

2. Створіть файл **`input.txt`**, який містить ваше прізвище, ім'я та по батькові. Відкрийте його для читання та виведіть інформацію на екран. Необхідно розглянути дві ситуації: коли файл знаходиться разом з програмою (тобто не треба вказувати шлях до файлу) і коли файл знаходиться в будь-якій іншій директорії. Крім того, слід передбачити повідомлення у разі, якщо файл не може бути відкритий.

3. Створіть три файли: **`input1.txt`**, **`input2.txt`** та **`input3.txt`**, в які занесіть відповідно ваше прізвище, ім'я та по батькові. Відкрийте файл **`output.txt`** для додавання та запишіть до нього в три рядки інформацію з трьох вхідних файлів.

4. Створіть бінарний файл **`bin.dat`** і запишіть до нього 16 випадкових цілих чисел, що складаються з двох цифр. Виведіть на екран вміст цього файлу, а також виведіть число, яке знаходиться у цьому файлі на тому місці, яке відповідає номеру вашого варіанта.

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:
   – тексти *C*–програм;
   – результати виконання *C*-програм.
4. Висновки по роботі.

Контрольні запитання:

1. Що таке файл?
2. У чому полягають особливості текстових файлів?
3. В чому перевага використання файлів у порівнянні з масивами?
4. Які операції можна проводити з файлами?
5. Яким чином відбувається запис до файлу?
6. Як відбувається читання з файлу?
7. Як отримати доступ до елементу файлу з заданим номером?
8. Як закрити файл та чому це потрібно робити?
9. Чи може файл складатися тільки з одного елемента?
10. Який файл називається текстовим?
11. Який файл називається бінарним?
12. Чи можуть рядки в текстових файлах мати різну довжину?
13. Яка максимальна довжина рядка в текстовому файлі?
14. Чи можна текстовий файл відкрити одночасно для читання та запису?

СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ

1. Code Blocks IDE – настройка и использование. Елена Завьялова – 27.11.2018. URL: https://gitjournal.tech/code-blocks-ide-nastrojka-i-ispolzovanie/ (дата обращения: 07.07.2021).

2. Code::Blocks / Downloads / Binary releases. URL: https://www.codeblocks.org/downloads/binaries/ (дата звернення: 07.07.2021).

3. ГОСТ 19.701–90 (ИСО 5807–85) ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения. – М.: Издательство стандартов, 1991.

4. Прата С. Язык программирования С. Лекции и упражнения – М.: Диалектика–Вильямс, 2015. – 928 с.

5. Дейтел П., Дейтел Х. С для программистов с введением в С11 – М.: ДМК Пресс, 2014. – 544 с.

6. Дейтел П., Дейтел Х. Как программировать на С, 7-е издание, 2014. – 1000 с.

7. Керниган Б., Ритчи Д. Язык программирования С – М.: Диалектика–Вильямс, 2015. – 288 с.

8. Шилдт Г. С: полное руководство, классическое издание – М.: "Диалектика–Вильямс", 2017. – 704 с.

9. Седжвик, Роберт. Алгоритмы на С++. : Пер. с англ. – М. : ООО "И.Д. Вильямс", 2016. – 1056 с. : ил. – Парал. тит. англ.

10. Указатель в языке Си. URL: https://prog-cpp.ru/c-pointers/ (дата обращения: 27.10.2019).

ДОДАТКИ

Додаток 1

Гарячі клавіші Code::Blocks

Таблиця Д 1.1 – Робота з файлами

Функція	Гарячі клавіші
Створити новий файл або проєкт	**Ctrl + N**
Відкрити існуючий файл або проєкт	**Ctrl + O**
Зберегти поточний файл	**Ctrl + S**
Зберегти всі файли	**Ctrl + Shift + S**
Закрити поточний файл	**Ctrl + F4 / Ctrl + W**
Закрити всі файли	**Ctrl + Shift + F4 / Ctrl + Shift + W**
Активувати наступний відкритий файл	**Ctrl + Tab**
Активувати попередній відкритий файл	**Ctrl + Shift + Tab**

Таблиця Д 1.2 – Робота з редактором

Функція	Гарячі клавіші
Відмінити останню дію	**Ctrl + Z**
Повторити останню дію	**Ctrl + Shift + Z**
Вирізати виділений текст	**Ctrl + X**
Копіювати виділений текст	**Ctrl + C**
Вставити текст з буфера обміну	**Ctrl + V**
Вибрати весь текст	**Ctrl + A**
Переключити текст і заголовки	**F11**
Коментувати виділений код	**Ctrl + Shift + C**
Розкоментувати виділений код	**Ctrl + Shift + X**
Дублікат рядку	**Ctrl + D**
Заповнити форму / Скорочення	**Ctrl +** пробіл **/ Ctrl + J**
Показати підказку	**Ctrl + Shift +** пробіл
Перемістити рядок вище	**Ctrl + T**
Переключити закладку	**Ctrl + B**
Перейти до попередньої закладки	**Alt + PgUp**
Перейти до наступної закладки	**Alt + PgDown**
Перемикання блокування поточного блоку	**F12**
Переключити всі вкладки	**Shift + F12**

Продовження додатка 1

Таблиця Д 1.3 – Робота з текстом

Функція	Гарячі клавіші
Збільшення розміру тексту	**Ctrl + Keypad "+"**
Зменшити розмір тексту	**Ctrl + Keypad "–"**
Відновити розмір тексту до нормального	**Ctrl + Keypad "/"**
Перебирати останні файли	**Ctrl + Tab**
Відступ вправо	**Tab**
Відступ вліво	**Shift + Tab**
Видалити до початку слова	**Ctrl + BackSpace**
Видалити до кінця слова	**Ctrl + Del**
Видалити до початку рядка	**Ctrl + Shift + BackSpace**
Видалити до кінця рядка	**Ctrl + Shift + Del**
На початок документа	**Ctrl + Home**
Розширення вибору для початку документа	**Ctrl + Shift + Home**
На початок рядка	**Alt + Home**
Збільшити вибір до початку рядка	**Alt + Shift + Home**
Перейти до кінця документа	**Ctrl + End**
Розширити виділення до кінця документа	**Ctrl + Shift + End**
Перейти до кінця рядка	**Alt + End**
Розширити виділення до кінця рядка	**Alt + Shift + End**
Розгорнути або згорнути блок	**Ctrl + Keypad + "*"**
Створити або видалити закладку	**Ctrl + B**
Перейти до наступної закладки	**Alt + PageDown**
Перейти до попередньої закладці	**Alt + PageUp**
Перейти до наступної зміненого рядка	**Ctrl + F3**
Перейти до попереднього зміненого рядка	**Ctrl + Shift + F3**
Прокрутити вгору	**Ctrl + Up**
Прокрутити вниз	**Ctrl + Down**
Вирізати рядок	**Ctrl + L**
Скопіювати рядок	**Ctrl + Shift + T**
Видалити рядок	**Ctrl + Shift + L**
Поміняти рядок з попереднім	**Ctrl + T**
Дублювати рядок	**Ctrl + D**
Попередній абзац. **Shift** розширює вибір.	**Ctrl + [**
Наступний параграф. **Shift** розширює вибір.	**Ctrl +]**
Попереднє слово. **Shift** розширює вибір.	**Ctrl + Left**
Наступне слово. **Shift** розширює вибір.	**Ctrl + Right**
Частина попереднього слова. **Shift** розширює вибір.	**Ctrl + /**
Наступна частина слова. **Shift** розширює вибір.	**Ctrl + **

Продовження додатка 1

Таблиця Д 1.4 – Перегляд

Функція	Гарячі клавіші
Показати / приховати панель повідомлень	**F2**
Показати / приховати панель управління	**Shift + F2**
Перемістити проєкт вгору (в дереві проєкту)	**Ctrl + Shift + Up**
Перенести проєкт вниз (в дереві проєкту)	**Ctrl + Shift + Down**
Активувати попередній (в дереві проєкту)	**Alt + F5**
Активувати наступний (в дереві проєкту)	**Alt + F6**
Збільшення / зменшення масштабу	**Ctrl +** колесо прокрутки миші
Редактор фокусування	**CTRL + Alt + E**

Таблиця Д 1.5 – Пошук

Функція	Гарячі клавіші
Знайти	**Ctrl + F**
Знайти наступне	**F3**
Знайти попереднє	**Shift + F3**
Знайти в файлах	**Crtl + Shift + F**
Замінити	**Ctrl + R**
Замінити в файлах	**Ctrl + Shift + R**
Перейти до рядка	**Ctrl + G**
Перейти до наступної змінної рядка	**Ctrl + F3**
Перейти до попередньої змінної рядка	**Ctrl + Shift + F3**
Перейти до файлу	**Alt + G**
Функція Goto	**Ctrl + Alt + G**
Перейти до попередньої функції	**Ctrl + PgUp**
Перейти до наступної функції	**Ctrl + PgDn**
Перейти до декларації функції	**Ctrl + Shift + "."**
Перейти до реалізації функції	**Ctrl + "."**
Відкрити файл include	**Ctrl + Alt + "."**

Таблиця Д 1.6 – Компонування

Функція	Гарячі клавіші
Компонувати (Build)	**Ctrl + F9**
Скомпілювати поточний файл (Compile current file)	**Ctrl + Shift + F9**
Запустити (Run)	**Ctrl + F10**
Компонувати і запустити (Build and run)	**F9**
Перекомпонувати (Rebuild)	**Ctrl + F11**

Продовження додатка 1

Таблиця Д 1.7 – Відлагодження

Функція	Гарячі клавіші
Почати відлагодження	**F8**
Продовжити відлагодження	**Ctrl + F7**
Пропустити блок коду	**F7**
Увійти до блоку коду	**Shift + F7**
Вийти з блоку коду	**Ctrl + Shift + F7**
Переключити точку зупину	**F5**
Виконати до курсору	**F4**
Попередня помилка	**Alt + F1**
Наступна помилка	**Alt + F2**

Додаток 2

Результати роботи для завдань 4 і 5 лабораторної роботи №5

Приклад результату виконання програми з визначення медіани для масиву випадкових чисел в діапазоні [20, 90] наведено на рис. Д 2.1.

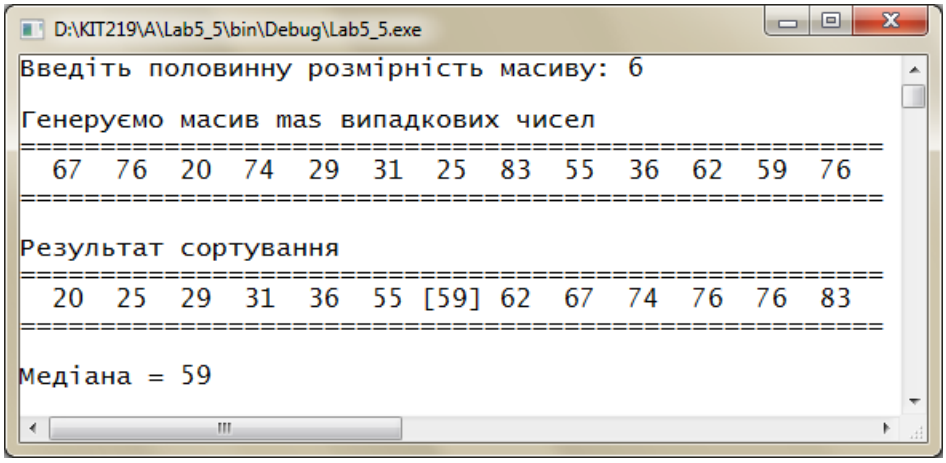


Рисунок Д 2.1 – Приклад результату роботи програми з визначення медіани для масиву випадкових чисел в діапазоні [20, 90]

Приклади результатів виконання програми з визначення моди (мод) для масиву випадкових чисел в діапазоні [0, 30] наведені на рис. Д 2.3 – Д 2.4.

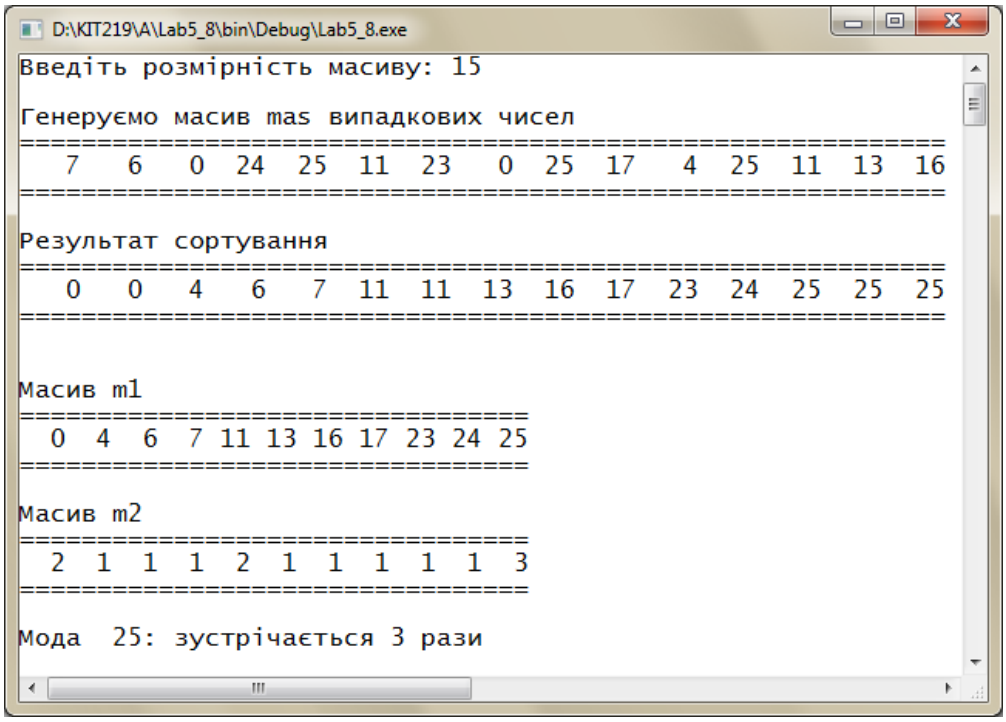


Рисунок Д 2.3 – Приклад результату роботи програми з визначення моди для масиву випадкових чисел в діапазоні [0, 30]

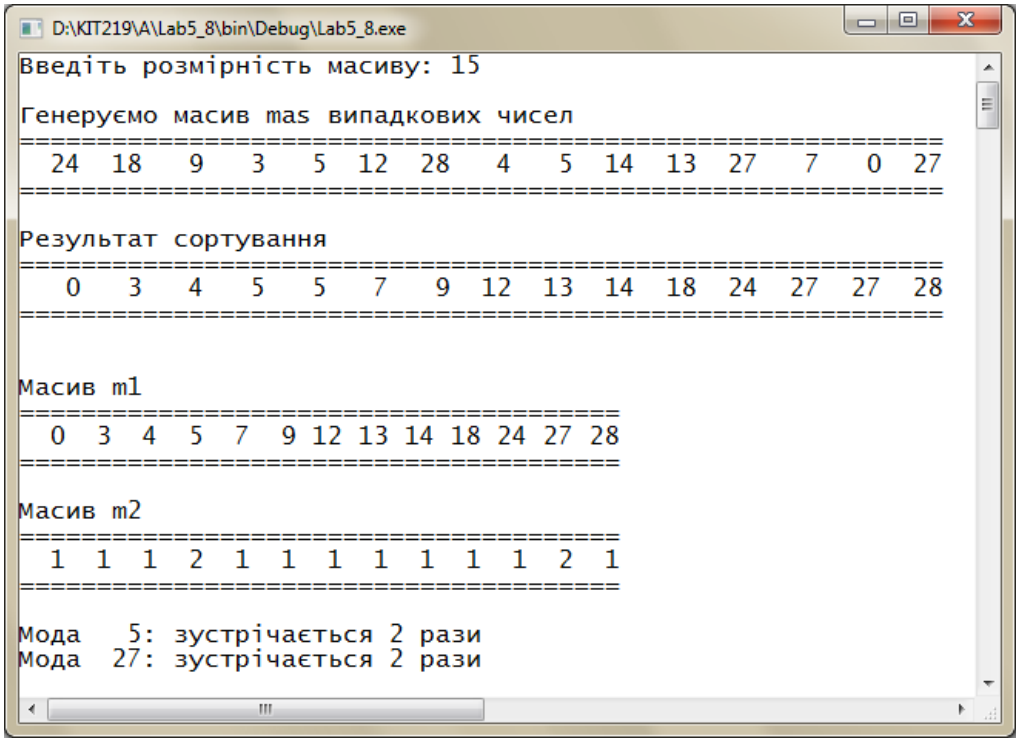


Рисунок Д 2.4 – Приклад результату роботи програми з визначення мод для масиву випадкових чисел в діапазоні [0, 30]

Додаток 3

Формули для визначення площ геометричних фігур

Трикутник:

1) Формула площі трикутника за стороною та висотою

$$S = \frac{1}{2}ah,$$

де a – сторона трикутника; h – висота трикутника, яка проведена до сторони a.

2) Формула площі трикутника за двома сторонами та кутом між ними

$$S = \frac{1}{2}ab\cdot\sin(\gamma),$$

де a, b – сторони трикутника; γ – значення кута між сторонами a і b.

3) Формула площі трикутника за трьома сторонами (формула Герона)

$$S = \sqrt{p(p-a)(p-b)(p-c)},$$

де a, b, c – сторони трикутника; p – півпериметр трикутника, що обчислюється за формулою $p = (a+b+c)/2$.

4) Формула площі трикутника за трьома сторонами та радіусом вписаного кола

$$S = pr,$$

де p – півпериметр трикутника; r – радіус вписаного кола.

5) Формула площі трикутника за трьома сторонами та радіусом описаного кола

$$S = \frac{abc}{4R},$$

де a, b, c – сторони трикутника; R – радіус описаного кола.

Рівносторонній трикутник:

6) Площа рівностороннього трикутника

$$S = \frac{a^2\sqrt{3}}{4},$$

де a – сторона рівностороннього трикутника.

7) Площа рівностороннього трикутника за радіусом описаного кола

$$S = \frac{3\sqrt{3}R^2}{4},$$

де R – радіус описаного кола;

8) Площа рівностороннього трикутника за радіусом вписаного кола

$$S = 3\sqrt{3}r^2,$$

де r – радіус вписаного кола.

9) Площа рівностороннього трикутника за висотою

$$S = \frac{\sqrt{3}h^2}{3},$$

де h – висота трикутника.

Прямокутний трикутник:

10) Формула площі прямокутного трикутника

$$S = \frac{1}{2}ab,$$

де a, b – катети прямокутного трикутника.

Квадрат:

11) Формула площі квадрата за довжиною сторони

$$S = a^2,$$

де a – сторона квадрату.

12) Формула площі квадрата за довжиною діагоналі

$$S = \frac{1}{2}d^2,$$

де d – діагональ.

13) Площа квадрата за радіусом описаного кола

$$S = 2R^2,$$

де R – радіус описаного кола.

14) Площа квадрата за радіусом вписаного кола

$$S = 4r^2,$$

де r – радіус вписаного кола.

Прямокутник:

15) Площа прямокутника

$$S = ab,$$

де a, b – сторони прямокутника.

16) Площа прямокутника за довжиною діагоналі

$$S = \frac{1}{2} d^2 \sin(\varphi),$$

де d – діагональ прямокутника; φ – значення кута між діагоналями.

Паралелограм:

17) Формула площі паралелограма за довжиною сторони та висотою

$$S = ah,$$

де a – сторона паралелограма; h – висота паралелограма.

18) Формула площі паралелограма за двома сторонами та кутом між ними

$$S = ab \cdot \sin(\alpha),$$

де a, b – сторони паралелограма; α – значення кута між сторонами a і b.

19) Формула площі паралелограма за діагоналями та кутом між ними

$$S = \frac{1}{2} d_1 d_2 \cdot \sin(\varphi),$$

де d_1, d_2 – діагоналі паралелограма; φ – значення кута між діагоналями паралелограма.

Ромб:

20) Формула площі ромба за довжиною сторони та кутом

$$S = a^2 \cdot \sin(\alpha),$$

де a – сторона ромба; α – значення кута між сторонами ромба.

21) Формула площі ромба за довжиною його діагоналей

$$S = \frac{d_1 d_2}{2},$$

де d_1, d_2 – діагоналі ромба.

Продовження додатка З

Трапеція:

22) Формула площі трапеції

$$S = \frac{(a+b)\cdot h}{2},$$

де a, b – довжини основ трапеції; h – висота трапеції.

23) Площа трапеції за довжиною середньої лінії та висотою

$$S = mh,$$

де m – середня лінія трапеції; h – висота трапеції.

Чотирикутник:

24) Площа довільного вписаного чотирикутника (формула Брахмагупти)

$$S = \sqrt{(p-a)(p-b)(p-c)(p-d)},$$

де a, b, c, d – сторони чотирикутника; p – півпериметр чотирикутника, що визначається за формулою $p = (a+b+c+d)/2$.

Правильний шестикутник:

25) Площа правильного шестикутника

$$S = \frac{3\sqrt{3}a^2}{2},$$

де a – сторона правильного шестикутника.

26) Площа правильного шестикутника за радіусом вписаного кола

$$S = 2\sqrt{3}r^2,$$

де r – радіус вписаного кола.

Правильний багатокутник:

27) Площа вписаного в коло правильного багатокутника

$$S = \frac{1}{2}R^2 n \cdot sin\left(\frac{360}{n}\right),$$

де R – радіус описаного кола навколо правильного багатокутника; n – число кутів або сторін.

28) Площа правильного багатокутника

$$S = \frac{nar}{2},$$

де r – радіус вписаного кола; a – сторона правильного багатокутника; n – число кутів або сторін.

Коло:

29) Площа кола

$$S = \pi R^2,$$

де R – радіус.

30) Площа сектора кола

$$S = \frac{Rl}{2},$$

де R – радіус; l – довжина дуги сектора.

31) Площа сектора кола через кут

$$S = \frac{\pi R^2 \alpha}{360},$$

де R – радіус; α – значення кута сектора.

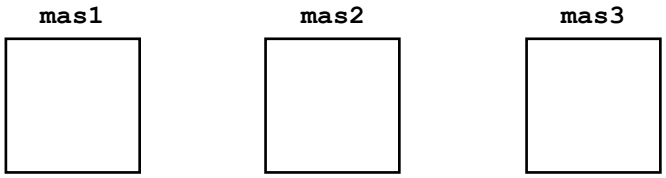
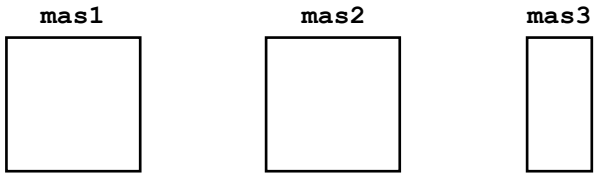
32) Площа сегмента кола

$$S = \frac{R^2}{2}\left(\frac{\pi\alpha}{180} - \sin(\alpha)\right),$$

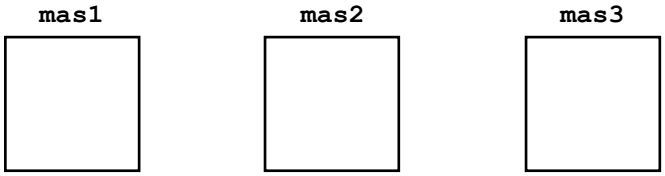
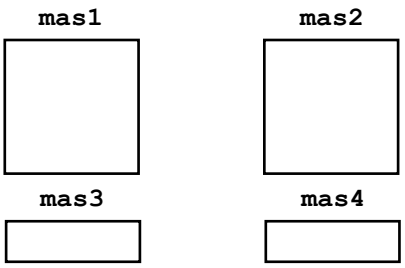
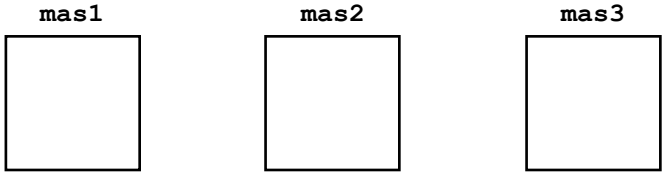
де R – радіус; α – значення кута дуги сегмента.

Додаток 4

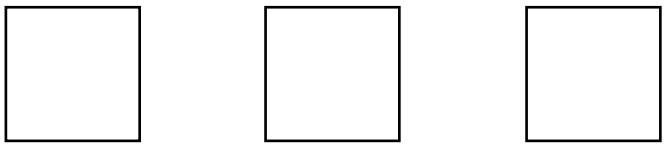
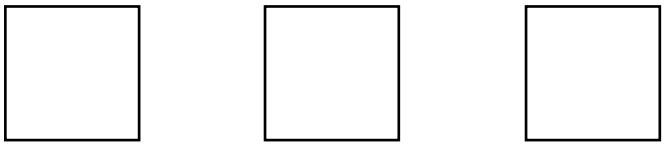
Варіанти завдання 3 лабораторної роботи №6

№ варіанта	Умови завдання
1	Функції, які необхідно реалізувати: 1) заповнення масиву розмірністю N×N випадковими цілими числами в діапазоні від 0 до 255 (функцію необхідно викликати в `main()` два рази для послідовного заповнення масивів **mas1** і **mas2**); 2) заповнення масиву **mas3** розмірністю N×N за принципом: – якщо **mas1[i][j]** і **mas2[i][j]** є парними, то до **mas3[i][j]** занести 0; – якщо **mas1[i][j]** і **mas2[i][j]** є непарними, то до **mas3[i][j]** занести 1; – якщо **mas1[i][j]** є парним, а **mas2[i][j]** – непарним, або навпаки, то до **mas3[i][j]** занести 2; 3) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним;  4) обчислення суми всіх елементів масиву **mas3**.
2	Функції, які необхідно реалізувати: 1) заповнення масиву **mas1** розмірністю N×N випадковими цілими числами від 0 до 255; 2) заповнення масиву **mas2** розмірністю N×N за принципом: – якщо **mas1[i][j]** є парним, то до **mas2[i][j]** занести 0; – якщо **mas1[i][j]** є непарним, то до **mas2[i][j]** занести 1; 3) заповнення одновимірного масиву **mas3** розмірністю N, в кожну комірку якого слід занести десяткове представлення двійкового коду, який розташований в рядках масиву **mas2**; 4) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним (масив **mas3** слід виводити вертикально);  5) обчислення суми всіх елементів масиву **mas3**.
3	Функції, які необхідно реалізувати: 1) заповнення масиву розмірністю N×N випадковими цілими числами в діапазоні від 0 до 255 (функцію необхідно викликати в `main()` два рази для послідовного заповнення двох масивів **mas1** і **mas2**); 2) проведення бульбашкового сортування рядків масиву **mas1** і стовпців масиву **mas2** за збільшенням; 3) вивід масивів **mas1** і **mas2** поруч один з одним (функцію необхідно викликати в main() два рази: після заповнення масивів **mas1** і **mas2** та після сортування); 5) заповнення масиву **mas3** розмірністю N×N за принципом:

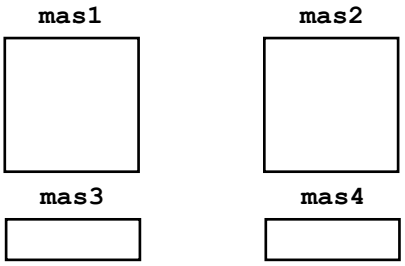
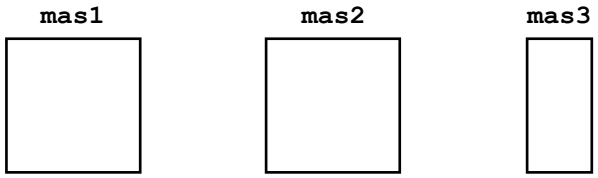
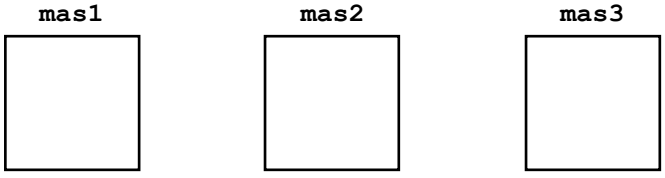
Продовження додатка 4

№ варіанта	Умови завдання
	`mas3[i][j] = mas1[i][j] + mas2[i][j];` 6) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним. 
4	Функції, які необхідно реалізувати: 1) заповнення масиву **mas1** розмірністю N×N (N – парне) випадковими цілими числами від 0 до 255; 2) заповнення масиву **mas2** розмірністю N×N за принципом: `mas2[0][0] = mas1[N-1][N-1], mas2[0][1] = mas1[N-1][N-2]` і т. д. (тобто останній елемент масиву **mas1** стає першим в масиві **mas2** і навпаки); 3) заповнення одновимірних масивів **mas3** і **mas4** розмірністю N сумою чисел, які розташовані в стовпцях масивів **mas1** і **mas2** відповідно; 4) вивід масивів **mas3** і **mas4** під масивами **mas1** і **mas2** відповідно. 
5	Функції, які необхідно реалізувати: 1) заповнення масиву **mas1** розмірністю N×N випадковими цілими числами від 0 до 255; 2) заповнення масиву **mas2** розмірністю N×N за принципом: `mas2[0][1]= 01; mas2[5][1] = 51;` і т. д.; 3) заповнення масиву **mas3** розмірністю N×N числами, кожне з яких є сумою одиниць у двійковому запису числа відповідної комірки масиву **mas1**; 4) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним. 
6	Функції, які необхідно реалізувати: 1) заповнення масиву **mas** розмірністю N×N нулями; 2) заповнення масиву **mas** розмірністю N×N цілими числами від 1 до N^2 випадковим чином. Тобто чергове число заносимо до комірки масиву, індекс якої визначається випадковим чином. Якщо комірка вже зайнята,

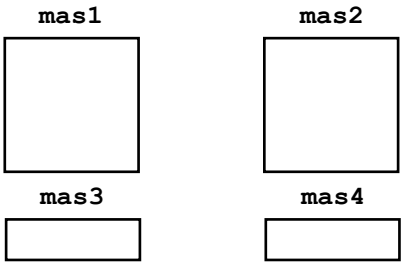
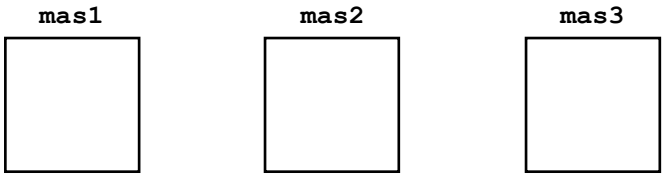
Продовження додатка 4

№ варіанта	Умови завдання
	визначається випадковим чином наступний індекс, доки отримана комірка не буде вільною (тобто не буде містити 0) і т. д.; 3) вивід квадратного двовимірного масиву **mas** на консоль. Розроблені функції слід викликати в циклі 6 разів, щоб було видно, що комірки в масиві **mas** заповнюються довільним чином.
7	Функції, які необхідно реалізувати: 1) заповнення масиву розмірністю N×N випадковими цілими числами в діапазоні від 0 до 255 (функцію необхідно викликати в `main()` два рази для послідовного заповнення двох масивів **mas1** і **mas2**); 2) заповнення масиву **mas3** розмірністю N×N за принципом: – якщо **i==j**, то **mas3[i][j] = mas1[i][j] & mas2[i][j]**, інакше: – якщо сума індексів **i+j** є парним числом, то **mas3[i][j] = mas1[i][j] ^ mas2[i][j];** – якщо сума індексів **i+j** є непарним числом, то **mas3[i][j] = mas1[i][j] | mas2[i][j];** 4) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним; **mas1** **mas2** **mas3**  4) обчислення суми всіх елементів масиву **mas3**, які діляться на 3 націло.
8	Функції, які необхідно реалізувати: 1) заповнення масиву розмірністю N×N випадковими цілими числами в діапазоні від 0 до 255 (функцію необхідно викликати в `main()` два рази для послідовного заповнення двох масивів **mas1** і **mas2**); 2) проведення бульбашкового сортування парних рядків масиву **mas1** і непарних стовпців масиву **mas2** за зменшенням; 3) вивід масивів **mas1** і **mas2** поруч один з одним (функцію необхідно викликати в `main()` два рази: після заповнення масивів **mas1** і **mas2** та після сортування); 4) заповнення масиву **mas3** розмірністю N×N за принципом: **mas3[i][j] = (mas1[i][j] + mas2[i][j]) >> 2;** 5) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним. **mas1** **mas2** **mas3** 
9	Функції, які необхідно реалізувати: 1) заповнення масиву **mas1** розмірністю N×N випадковими цілими числами в діапазоні від 0 до 255; 2) заповнення масиву **mas2** розмірністю N×N за принципом: – парні рядки заповнити відповідними парними рядками масиву **mas1** у зворотному порядку;

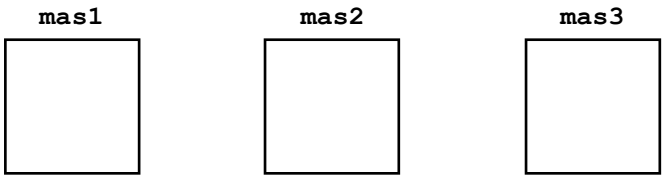
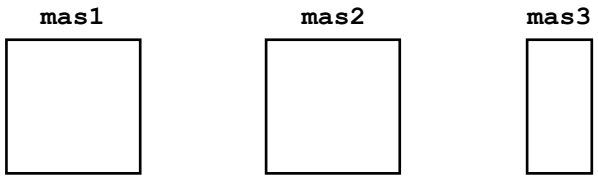
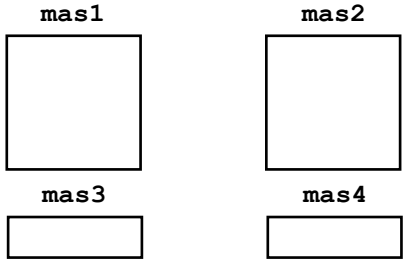
Продовження додатка 4

№ варіанта	Умови завдання
	– непарні рядки заповнити відповідними непарними рядками масиву **mas1** у прямому порядку; 3) вивід масивів **mas1** і **mas2** поруч один з одним; 4) заповнення одновимірних масивів **mas3** і **mas4** розмірністю N сумами чисел, які розташовані в стовпцях масивів **mas1** і **mas2** відповідно; 5) вивід масивів **mas3** і **mas4** під масивами **mas1** і **mas2** відповідно. 
10	Функції, які необхідно реалізувати: 1) заповнення парних стовпців масиву **mas1** розмірністю N×N (N – непарне) випадковими цілими числами від 1 до 255 та заповнення інших елементів масиву 0; 2) заповнення масиву **mas2** розмірністю N×N за принципом: – непарні стовпці масиву **mas2** заповнити сумою елементів сусідніх стовпців (якщо сума перевищує 255, то слід занести до комірки число 255); – парні стовпці **mas2** залишити такими, як у масиві **mas1**; 3) заповнення одновимірного масиву **mas3** розмірністю N, в кожну комірку якого слід занести суму елементів, що розташовані в рядках масиву **mas2**; 4) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним (масив **mas3** слід виводити вертикально);  5) обчислення суми всіх елементів масиву **mas3**.
11	Функції, які необхідно реалізувати: 1) заповнення масиву **mas1** розмірністю N×N випадковими цілими числами від 0 до 255; 2) заповнення масиву **mas2** розмірністю N×N за принципом: `mas2[0][1] = 0 * 1; mas2[3][3] = 3 * 3;` і т. д.; 3) заповнення масиву **mas3** розмірністю N×N числами, кожне з яких є сумою нулів у двійковому запису числа відповідної комірки масиву **mas1**; 4) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним. 

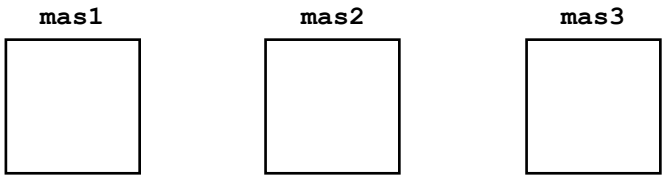
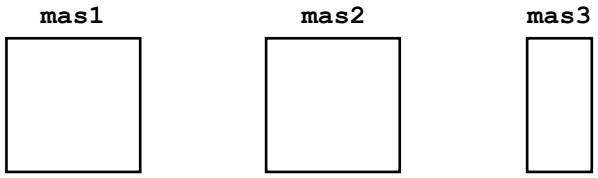
Продовження додатка 4

№ варіанта	Умови завдання
12	Функції, які необхідно реалізувати: 1) заповнення масиву **mas1** розмірністю N×N випадковими цілими числами від 0 до 255; 2) заповнення масиву **mas2** розмірністю N×N за принципом: рядок масиву **mas2** заповнити стовпцем масиву **mas1**; 3) вивід масивів **mas1** і **mas2** поруч один з одним; 4) заповнення одновимірних масивів **mas3** і **mas4** розмірністю N сумами чисел, які розташовані в стовпцях масивів **mas1** і **mas2** відповідно; 5) вивід масивів **mas3** і **mas4** під масивами **mas1** і **mas2** відповідно. 
13	Функції, які необхідно реалізувати: 1) заповнення двох рядків масиву **mas1** розмірністю N×N випадковими цілими числами від 1 до 5 та заповнення інших елементів масиву 0; 2) заповнення масиву **mas2** розмірністю N×N за принципом: – перші два рядки масиву **mas2** повинні співпадати з відповідними рядками масиву **mas1**; – кожен наступний рядок масиву **mas2** необхідно заповнити сумою елементів поточного стовпця, які розташовані на двох попередніх рядках; 3) заповнення масиву **mas3** розмірністю N×N числами, які є сумою відповідних елементів масивів **mas1** і **mas2**. Якщо отриманий результат після додавання більше 255, до комірки слід занести число, яке необхідно спочатку зсунути на 2 розряди праворуч, тобто: `mas3[i][j] = (mas1[i][j] + mas2[i][j]) >> 2;` 4) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним. 
14	Функції, які необхідно реалізувати: 1) заповнення масиву розмірністю N×N випадковими цілими числами від –15 до 15 (функцію необхідно викликати в `main()` два рази для послідовного заповнення двох масивів **mas1** і **mas2**); 2) проведення бульбашкового сортування рядків масиву **mas1** та стовпців масиву **mas2** за зростанням; 3) вивід масивів **mas1** і **mas2** поруч один з одним (функцію необхідно викликати в `main()` два рази: після заповнення масивів **mas1** і **mas2** та після сортування); 4) заповнення третього масиву **mas3** розмірністю N×N за принципом:

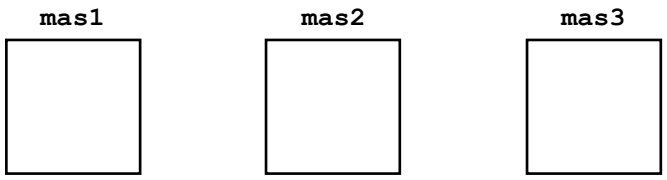
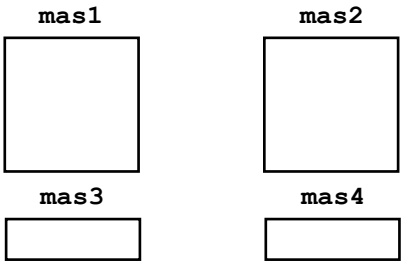
Продовження додатка 4

№ варіанта	Умови завдання
	`mas3[i][j] = mas1[i][j] + mas2[i][j];` 5) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним. 
15	Функції, які необхідно реалізувати: 1) заповнення масиву **mas1** розмірністю N×N випадковими цілими числами від –125 до 125; 2) заповнення масиву **mas2** розмірністю N×N за принципом: – якщо **mas1[i][j]** є парним, то до **mas2[i][j]** занести 0; – якщо **mas1[i][j]** є непарним, то до **ma2[i][j]** занести 1; 3) заповнення одновимірного масиву **mas3** розмірністю N, в кожну комірку якого слід занести десяткове представлення двійкового коду, який розташований в рядках масиву **mas2**; 4) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним (масив **mas3** слід виводити вертикально);  5) обчислення суми всіх елементів масиву **mas3**.
16	Функції, які необхідно реалізувати: 1) заповнення масиву **mas1** розмірністю N×N випадковими цілими числами від 0 до 255; 2) заповнити масив **mas2** розмірністю N×N за принципом: – непарні рядки заповнити відповідними непарними рядками масиву **mas1** у зворотному порядку; – парні рядки заповнити відповідними парними рядками масиву **mas1** у прямому порядку; 3) вивід масивів **mas1** і **mas2** поруч один з одним; 4) заповнення одновимірних масивів **mas3** і **mas4** розмірністю N сумами чисел, які розташовані в стовпцях масивів **mas1** і **mas2** відповідно; 5) вивід масивів **mas3** і **mas4** під масивами **mas1** і **mas2** відповідно. 

Продовження додатка 4

№ варіанта	Умови завдання
17	Функції, які необхідно реалізувати: 1) заповнення масиву розмірністю N×N випадковими цілими числами в діапазоні від 50 до 205 (функцію необхідно викликати в `main()` два рази для послідовного заповнення масивів **mas1** і **mas2**); 2) заповнення масиву **mas3** розмірністю N×N за принципом: – якщо **mas1[i][j]** і **mas2[i][j]** є парними, то до **mas3[i][j]** занести 0; – якщо **mas1[i][j]** і **mas2[i][j]** є непарними, то до **mas3[i][j]** занести 1; – якщо **mas1[i][j]** є парним, а **mas2[i][j]** – непарним, або навпаки, то до **mas3[i][j]** занести 3; 3) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним;  4) обчислення кількості елементів масиву **mas3**, які містять 0, 1 і 3.
18	Функції, які необхідно реалізувати: 1) заповнення масиву **mas1** розмірністю N×N випадковими цілими числами від 10 до 199; 2) заповнення масиву **mas2** розмірністю N×N за принципом: – якщо **mas1[i][j]** є парним, то до **mas2[i][j]** занести 0; – якщо **mas1[i][j]** є непарним, то до **mas2[i][j]** занести 1; 3) заповнення одновимірного масиву **mas3** розмірністю N, в кожну комірку якого слід занести десяткове представлення двійкового коду, який розташований в рядках масиву **mas2**; 4) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним (масив **mas3** слід виводити вертикально);  5) обчислення суми всіх елементів масиву **mas3**.
19	Функції, які необхідно реалізувати: 1) заповнення масиву розмірністю N×N випадковими цілими числами в діапазоні від 30 до 70 (функцію необхідно викликати в main() два рази для послідовного заповнення двох масивів **mas1** і **mas2**); 2) проведення бульбашкового сортування **рядків** масиву **mas1** і стовпців масиву **mas2** за збільшенням; 3) вивід масивів **mas1** і **mas2** поруч один з одним (функцію необхідно викликати в `main()` два рази: після заповнення масивів **mas1** і **mas2** та після сортування); 5) заповнення масиву **mas3** розмірністю N×N за принципом: `mas3[i][j] = mas1[i][j] – mas2[i][j];` 6) вивід масивів **mas1**, **mas2** та **mas3** поруч один з одним.

Продовження додатка 4

№ варіанта	Умови завдання
	
20	Функції, які необхідно реалізувати: 1) заповнення масиву **mas1** розмірністю N×N (N – парне) випадковими цілими числами від 16 до 155; 2) заповнення масиву **mas2** розмірністю N×N за принципом: **mas2[0][0] = mas1[N-1][N-1], mas2[0][1] = mas1[N-1][N-2]** і т. д. (тобто останній елемент масиву **mas1** стає першим в масиві **mas2** і навпаки); 3) заповнення одновимірних масивів **mas3** і **mas4** розмірністю N сумою чисел, які розташовані в стовпцях масивів **mas1** і **mas2** відповідно; 4) вивід масивів **mas3** і **mas4** під масивами **mas1** і **mas2** відповідно. 

Додаток 5

Приклад програми для роботи з функціями

```
//============================================================
// Приклад програми, яка викликає функцію для заповнення
// двовимірного масиву NxN зі змінною довжиною випадковими
// числами. Двовимірний масив у програмі використовується як
// одновимірний, але виводиться як двовимірний (N = 5, 7, 9).
//============================================================
#include <stdio.h>
#include <windows.h>
#include <time.h>
//============================================================
// Функція для ініціалізації двовимірного масиву NxN
// випадковими числами з діапазону від 10 до 88
//============================================================
void init_mas(int mas[], int n)
{
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
      mas[i * n + j] = rand() % (88 - 10 + 1) + 10;
}
//============================================================
// Функція для виводу на консоль масиву NxN
//============================================================
void print_mas(int mas[], int n)
{
  printf("\nМасив mas:\n");
  for(int k = 0; k < n; k++) printf("====");
  printf("\n\n");
  for(int i = 0; i < n; i++)
  {
    for(int j = 0; j < n; j++) printf("%4d", mas[i * n + j]);
      printf("\n\n");
  }
  for(int k = 0; k < n; k++) printf("====");
  printf("\n\n");
}
int main(void)
{
  int n;
  SetConsoleOutputCP(1251);
  printf("Введіть розмірність квадратного двовимірного масиву:");
  scanf("%d", &n);
  int mas[n * n];         // Виділяємо пам'ять під масив
  srand(time(NULL));
  init_mas(mas, n);       // Заповнюємо масив випадковими числами
  print_mas(mas, n);      // Виводимо масив на консоль
  return 0;
}
```

Продовження додатка 5

Результати виконання програми для N = 7 і N = 9 наведені на рис. Д 5.1.

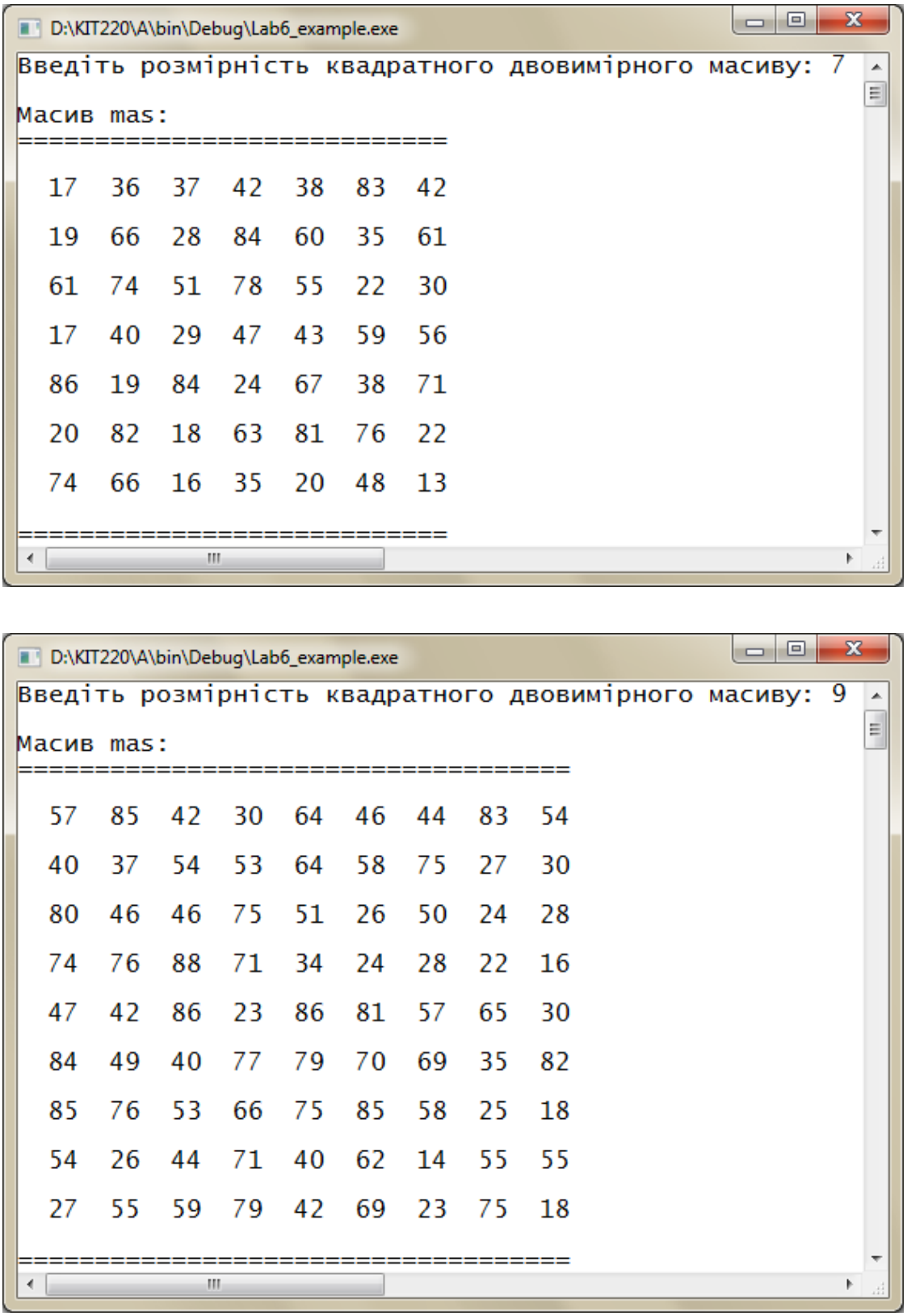


Рисунок Д 5.1 – Результати виконання програми для N = 7 і N = 9

Навчальне видання

Методичні вказівки
до виконання лабораторних робіт з дисципліни

«Програмування»

для студентів денної та заочної форм навчання за спеціальністю
123 «Комп'ютерна інженерія»
Частина 1

Укладачі: ПОРОШИН Сергій Михайлович
СТАТКУС Андрій Віталійович
НОСИК Андрій Михайлович
ОНИЩЕНКО Валерій Валентинович

Відповідальний за випуск проф. Порошин С. М.
Роботу рекомендував до друку проф. Заполовський М. Й.

В авторській редакції

План 2021 р., поз. 259

Підп. до друку 6.12.2021 р. Формат 60×84 1/16. Папір офсетний.
RISO-друк. Гарнітура Тайм
Ум. друк. арк. 9.2
Наклад 50 прим. Замовлення № . Ціна договірна

Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2

Видавництво та друк ФО-П Єфименко С.А.
61166, Україна, м. Харків, вул. Коломенська, 27
Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців – виготовлювачів і розповсюджувачів видавничої продукції
ДК № 6869 від 08.08.2019 р.