

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

С. В. Вишневий

ІНФОРМАТИКА. ЧАСТИНА 1. ОСНОВИ ПРОГРАМУВАННЯ ТА АЛГОРИТМИ КУРС ЛЕКЦІЙ

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітніми програмами «Інтелектуальні технології радіоелектронної техніки»,
«Інформаційна та комунікаційна радіоінженерія»,
«Радіотехнічні комп’ютеризовані системи»,
спеціальності
172 Електронні комунікації та радіотехніка

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2023

Рецензенти Сушко І.О., канд. техн. наук, доц. кафедри прикладної
радіоелектроніки, РТФ, Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Відповіdalnyj
редактор Жук С.Я., докт. техн. наук, професор

*Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 8 від 02.06.2023 р.)
за поданням Вченої ради Радіотехнічного факультету
(протокол № 05/2023 від 28.04.2023 р.)*

Навчальний посібник містить матеріали тем, які розглядаються в рамках лекційних занять в ході вивчення дисципліни «Інформатика. Частина 1. Основи програмування та алгоритми». Детально розглянуто системи числення та методики перетворення, надано інформацію щодо синтаксису мови програмування С, розглянуто принципи використання операторів та операцій мови програмування С, надано інформацію щодо особливостей роботи із одновимірними та багатовимірними статичними та динамічними масивами, приділено увагу питанням, що стосуються обробки символічних рядків, розглянуто особливості використання бібліотечних функцій та принципи написання власних функцій, надані приклади роботи із файлами, вказівниками, структурами та ін.

Для студентів першого року навчання радіотехнічного факультету, що проходять навчання за спеціальністю 172 «Електронні комунікації та радіотехніка».

Реєстр. № 22/23-726. Обсяг 10 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
<https://kpi.ua>

Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ЗМІСТ

ВСТУП.....	6
1. ІНТЕГРОВАНЕ СЕРЕДОВИЩЕ РОЗРОБКИ CODE::BLOCKS ДЛЯ НАПИСАННЯ ПРОГРАМ МОВОЮ С	7
2. СИСТЕМИ ЧИСЛЕННЯ.....	19
2.1 Двійкова система числення	22
2.2 Вісімкова система числення.....	35
2.3 Шістнадцяткова система числення	38
2.4 Методика швидкого перетворення двійкового коду у вісімковий код або шістнадцятковий код. Зворотне перетворення	41
2.5 Пряний код. Представлення беззнакових та додатних цілих чисел в пам'яті комп'ютера	48
2.6 Доповняльний код. Представлення від'ємних цілих чисел в пам'яті комп'ютера.....	53
2.7 Формат з фіксованою комою	57
2.8 Формат з плаваючою комою для зберігання дійсних чисел.....	62
2.9 Питання для самоконтролю	67
3. СТРУКТУРА ПРОГРАМИ НА МОВІ С. ТИПИ ДАНИХ. ОПЕРАЦІЇ	68
3.1 Структура програми на мові програмування С	68
3.2 Типи даних мови програмування С.....	74
3.3 Операції мови програмування С.....	100
3.4 Питання для самоконтролю	120
4. УМОВНІ ОПЕРАТОРИ	121
4.1 Умовний оператор вибору if.....	121
4.2 Умовний оператор вибору if/else.....	131

4.3 Оператор switch	141
4.4 Оператор циклу for	148
4.5 Оператор циклу while	152
4.6 Оператор циклу do/while	154
4.7 Оператори continue та break та їх використання в операторах циклу.....	157
4.8 Питання для самоконтролю	160
5. ВКАЗІВНИКИ. ОСНОВИ РОБОТИ ІЗ ФУНКЦІЯМИ	161
5.1 Особливості використання вказівників	162
5.2 Вказівник на вказівник	178
5.3 Основи роботи із функціями.....	184
5.4 Питання для самоконтролю	201
6. ОДНОВИМІРНІ ТА БАГАТОВИМІРНІ СТАТИЧНІ МАСИВИ.....	202
6.1 Одновимірний статичний масив. Принципи роботи із одновимірними статичними масивами	202
6.2 Багатовимірний статичний масив. Принципи роботи із багатовимірними статичними масивами на прикладі двовимірного масиву	211
6.3 Питання для самоконтролю	227
7. ОДНОВИМІРНІ ТА БАГАТОВИМІРНІ ДИНАМІЧНІ МАСИВИ.....	228
7.1 Одновимірний динамічний масив. Особливості роботи із одновимірними динамічними масивами	231
7.2 Багатовимірні динамічні масиви. Особливості роботи із багатовимірними динамічними масивами	240
7.3 Питання для самоконтролю	254

8. РОБОТА З ФАЙЛАМИ. СИМВОЛЬНІ РЯДКИ ТА ФУНКЦІЇ ПО РОБОТІ ІЗ СИМВОЛЬНИМИ РЯДКАМИ	255
8.1. Особливості роботи із файлами. Види файлів	256
8.2. Функції по роботі із символними рядками.....	275
8.3. Питання для самоконтролю	282
9. СТРУКТУРИ. ОСНОВИ РОБОТИ ІЗ СТРУКТУРАМИ.....	283
9.1. Оголошення змінних структурного типу даних	284
9.2. Масиви структур та вказівники на структури.....	287
9.3. Принципи написання функцій по обробці структур	307
9.4. Питання для самоконтролю	314
СПИСОК ЛІТЕРАТУРИ.....	315

ВСТУП

Сучасні технічні рішення в галузі електронних комунікацій та радіотехніки, а також в інших сферах інформаційних технологій, потребують використання різноманітних методів обробки даних шляхом реалізації відповідних алгоритмів. Реалізація алгоритмів передбачає написання програм. Мова програмування С дозволяє створювати програми не тільки для персональних комп’ютерів, але і для цілого ряду мікроконтролерів, які знаходять своє застосування при розробці та виготовленні вбудованих систем та спеціалізованих пристройів різного призначення. Мова С характеризується високою швидкодією написаного коду, гарними можливостями по переносу розроблених рішень на різні аппаратні платформи, наявністю широкого спектру засобів для реалізації різноманітних алгоритмів в процедурній парадигмі.

В посібнику розглянуто особливості різних систем числення, які широко застосовуються в цифровій техніці, приведено методики перетворення у відповідні системи числення, розглянуто формати з фіксованою та плаваючою комою. Розглянуто синтаксис мови С, а також теми, які охоплюють наступні питання: операції мови С, умовні оператори, робота із функціями, принципи використання вказівників, особливості роботи із статичними та динамічними одновимірними та багатовимірними масивами, символними рядками, файлами, структурами. При написанні прикладів програм використовувалось середовище розробки Code::Blocks версії 20.03. Приведені результати виконання програм отримані при їх запуску на персональному комп’ютері під управлінням операційної системи Windows 7 Professional 64 bit.

Посібник призначений для студентів першого року навчання, і дозволяє сформувати базис для подальшого вивчення дисциплін, які пов’язані із програмуванням, зокрема, дисциплін, які стосуються програмування мікроконтролерів та вбудованих систем, а також може бути використаний для формування необхідних навичок та компетенцій, які будуть необхідні при самостійному вивчення інших мов програмування.

1. ІНТЕГРОВАНЕ СЕРЕДОВИЩЕ РОЗРОБКИ CODE::BLOCKS ДЛЯ НАПИСАННЯ ПРОГРАМ МОВОЮ С

Написання програм мовою програмування С потребує використання відповідного середовища розробки, а також необхідності встановлення на персональному комп’ютері відповідного компілятора. Одним із таким засобів для розробки є інтегроване середовище Code::Blocks. В разі неможливості встановити на персональний комп’ютер Code::Blocks, можуть бути використані інші середовища розробки.

Для встановлення інтегрованого середовища розробки Code::Blocks необхідно мати підключення до мережі Інтернет для завантаження інсталяційного файлу. Щоб завантажити файл для встановлення необхідно у вікні браузера перейти на сайт за посиланням <https://www.codeblocks.org/>. Приклад головного вікна сайту показано на рис. 1.1.



Рис. 1.1 — Приклад головної сторінки проекту CodeBlocks.org

Для завантаження інсталяційного файлу необхідно перейти на відповідну сторінку веб-сайту <https://www.codeblocks.org/> шляхом натискання лівою клавішею миші на пункт меню Downloads (рис. 1.2).



Рис. 1.2 — Перехід на сторінку веб-сайту для завантаження інсталяційного файла інтегрованого середовища розробки Code::Blocks

Перейшовши на сторінку веб-сайту <https://www.codeblocks.org/downloads> можна обрати варіант завантаження, наприклад, обрати пункт Download the binary release.

Приклад вікна сторінки <https://www.codeblocks.org/downloads/> із виділеним червоним прямокутником варіанту вибору, на який необхідно натиснути лівою клавішею миші, показано на рис. 1.3. Перехід за вказаним посиланням дозволяє відобразити у вікні браузера веб-сторінку із вибором завантаження інсталяційного файла для відповідної операційної системи. Потрібно пересвідчитися яка саме операційна система встановлена на комп’ютері, на який буде встановлюватися Code::Blocks, звертаючи увагу на

роздрібність операційної системи, та обрати відповідний файл для встановлення.

Приклад веб-сторінки із вибором операційної системи показано на рис. 1.4.



Рис. 1.3 — Приклад обрання варіанту для завантаження інсталяційного файлу Code::Blocks

[Code::Blocks / Downloads / Binary releases](#)

Binary releases

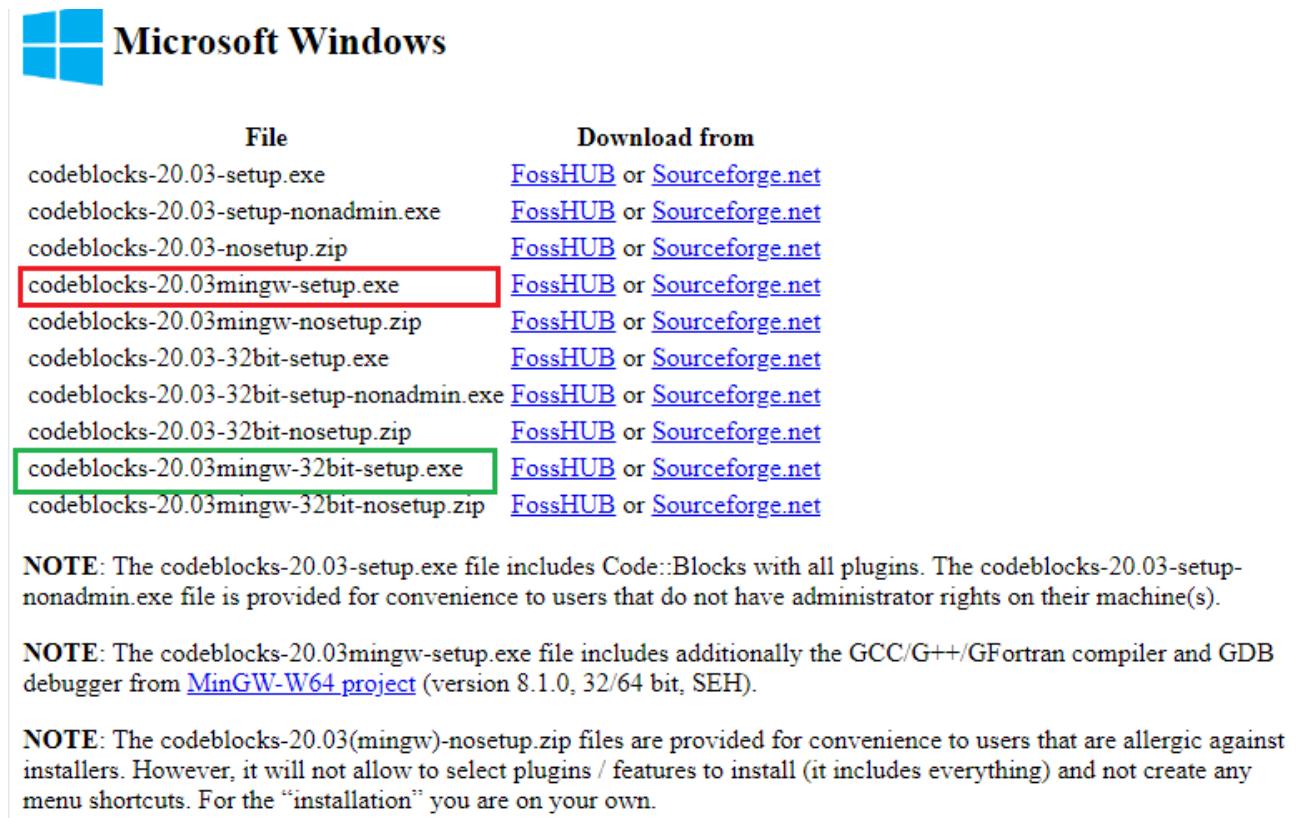
Please select a setup package depending on your platform:

- [Windows XP / Vista / 7 / 8.x / 10](#)
- [Linux 32 and 64-bit](#)
- [Mac OS X](#)

NOTE: For older OS'es use older releases. There are releases for many OS version and platforms on the [Sourceforge.net](#) page.

Рис. 1.4 — Сторінка обрання операційної системи, для подальшого завантаження відповідного інсталяційного файлу Code::Blocks

Наприклад, при встановленні інтегрованого середовища Code::Blocks на персональний комп'ютер, що працює під управлінням операційної системи Windows XP/Vista/7/8.x/10, при натисканні лівою кнопкою миші на відповідне посилання, що представлене на веб-сторінці (рис. 1.4), з'являються актуальні на момент встановлення варіанти інсталяційних файлів (рис. 1.5). Наприклад, для 64-х розрядної операційної системи сімейства Windows доцільно вибирати інсталяційний файл codeblocks-XX.XXmingw-setup.exe, а для 32-х розрядної операційної системи - codeblocks-XX.XXmingw-32-bit-setup.exe, де XX.XX — номер актуальної версії файлу на момент встановлення Code::Blocks. Щоб завантажити обраний варіант інсталяційного файлу необхідно натиснути на відповідне джерело завантаження FossHub або Sourceforge.net (або інше джерело, що буде відображатися на даній сторінці напроти відповідного файла на момент встановлення Code::Blocks).



File	Download from
codeblocks-20.03-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03-setup-nonadmin.exe	FossHUB or Sourceforge.net
codeblocks-20.03-nosetup.zip	FossHUB or Sourceforge.net
codeblocks-20.03mingw-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03mingw-nosetup.zip	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-setup-nonadmin.exe	FossHUB or Sourceforge.net
codeblocks-20.03-32bit-nosetup.zip	FossHUB or Sourceforge.net
codeblocks-20.03mingw-32bit-setup.exe	FossHUB or Sourceforge.net
codeblocks-20.03mingw-32bit-nosetup.zip	FossHUB or Sourceforge.net

NOTE: The codeblocks-20.03-setup.exe file includes Code::Blocks with all plugins. The codeblocks-20.03-setup-nonadmin.exe file is provided for convenience to users that do not have administrator rights on their machine(s).

NOTE: The codeblocks-20.03mingw-setup.exe file includes additionally the GCC/G++/GFortran compiler and GDB debugger from [MinGW-W64 project](#) (version 8.1.0, 32/64 bit, SEH).

NOTE: The codeblocks-20.03(mingw)-nosetup.zip files are provided for convenience to users that are allergic against installers. However, it will not allow to select plugins / features to install (it includes everything) and not create any menu shortcuts. For the “installation” you are on your own.

Рис. 1.5 — Обрання файла для встановлення для 64-х розрядної або 32-х розрядної операційної системи сімейства Windows

Після встановлення на персональний комп'ютер Code::Blocks та його запуску, з'являється наступне вікно (в подальшому буде показано вигляд відповідних вікон Code::Blocks версії 20.03 для 64-х розрядної операційної системи сімейства Windows), яке демонструється на рис. 1.6. Для створення нового проекту необхідно обрати варіант вибору Create a new project або обрати пункт меню File → New → Project... Після цього з'явиться вікно вибору типу проекту, яке показано на рис. 1.7.

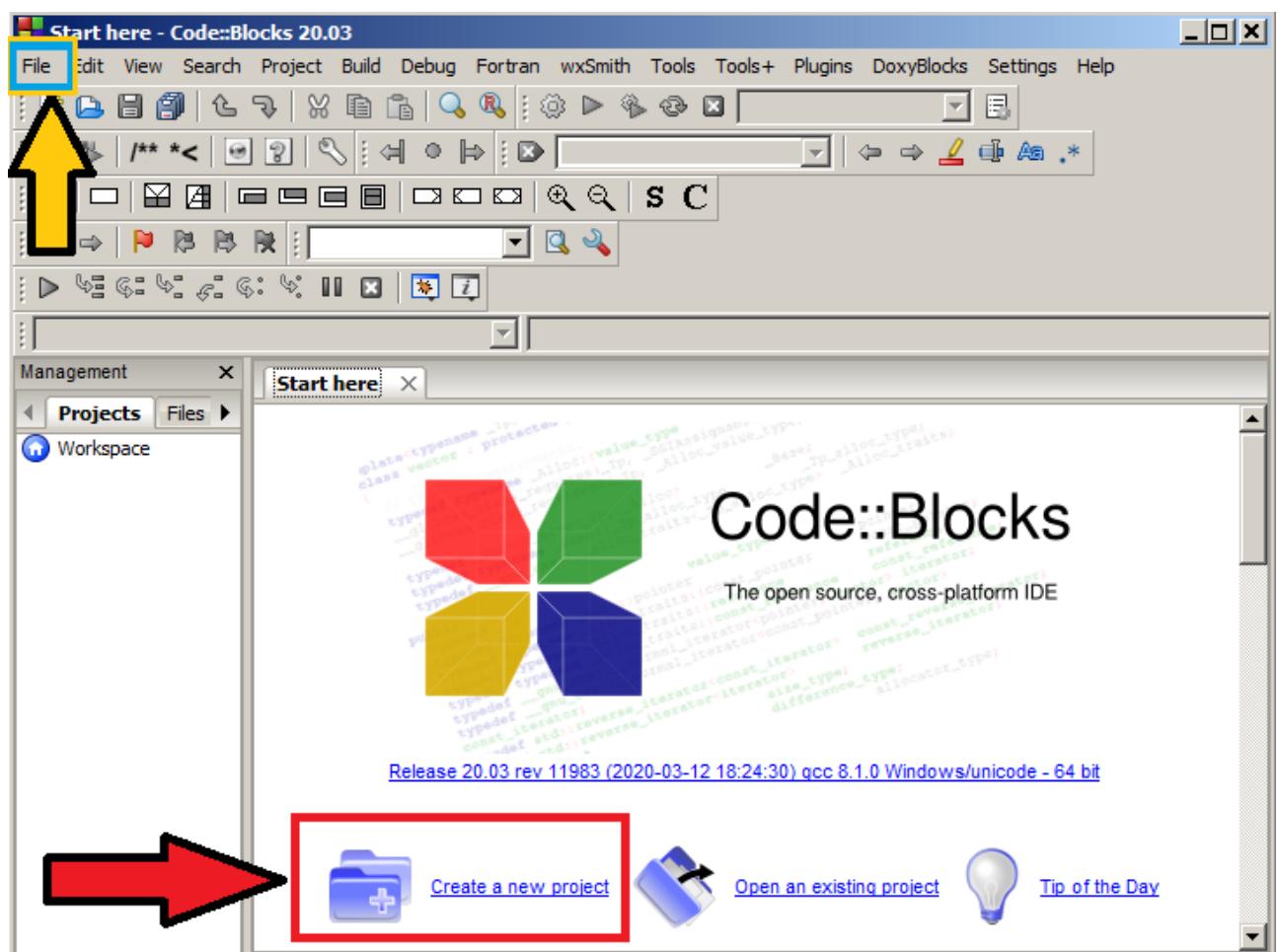


Рис. 1.6 — Створення нового проекту в інтегрованому середовищі розробки Code::Blocks

При написанні програм на мові програмування C передбачається створення проекту, який буде працювати в консольному вікні операційної системи. Таким чином, у діалоговому вікні, яке показано на рис. 1.7, необхідно вибрати пункт Console application.

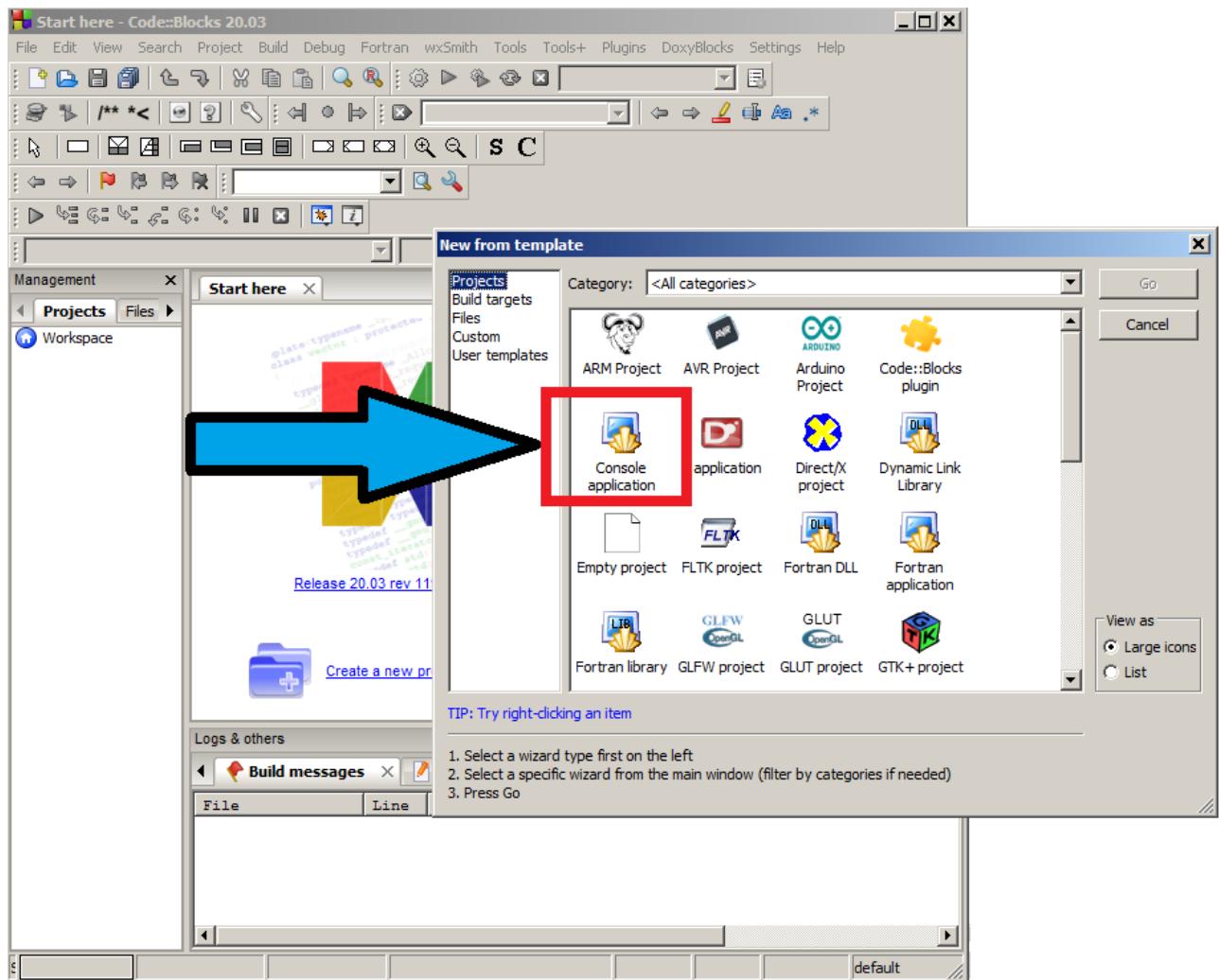


Рис. 1.7 — Вибір проекту Console application

Натиснувши лівою кнопкою миші на піктограмі Console application, і після цього натиснувши на кнопку Go, з'являється наступне діалогове вікно, що показано на рис. 1.8.

В діалоговому вікні на рис. 1.8 потрібно натиснути на кнопку Next, щоб перейти до наступного вікна створення проекту.

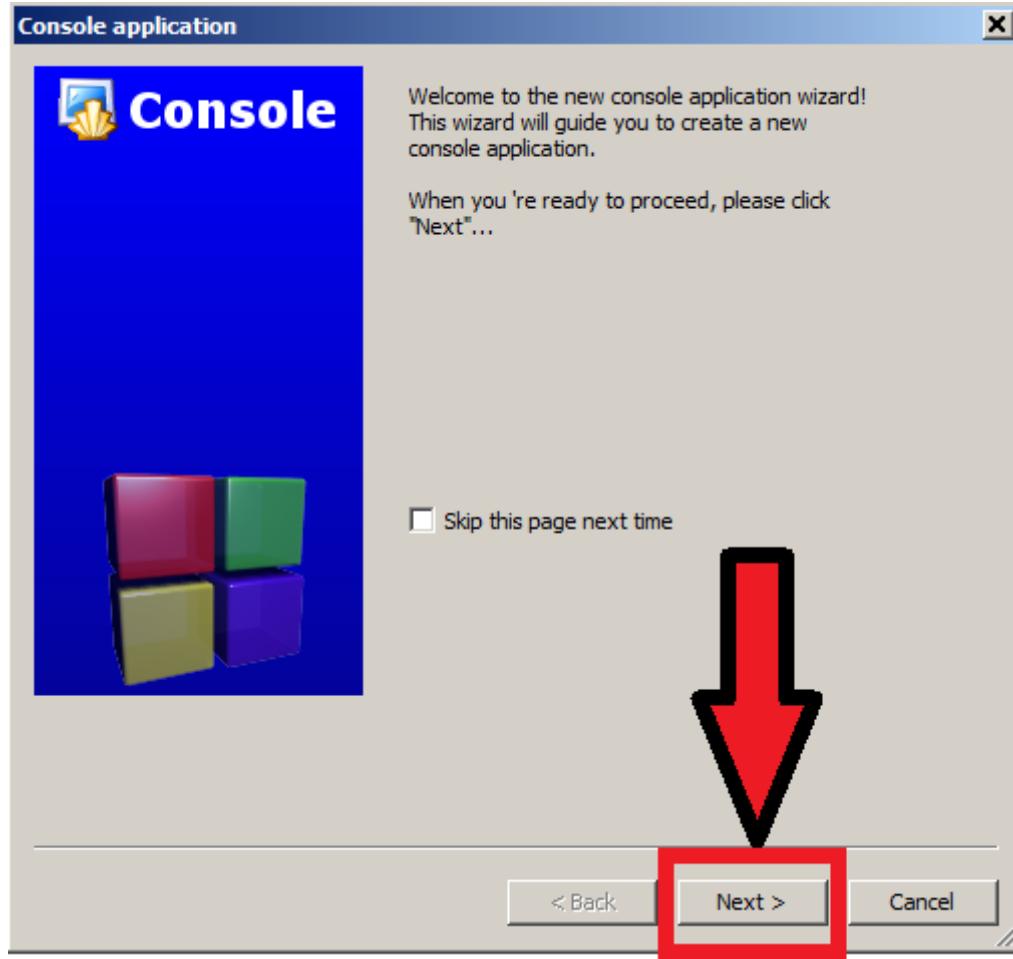


Рис. 1.8 — Натисканням кнопки Next переходимо до наступного діалогова вікна створення проекту

Наступне діалогове вікно створення проекту, яке виникає після натиснення кнопки Next в діалогову вікні на рис. 1.8, показано на рис. 1.9.

Діалогове вікно на рис. 1.9 передбачає вибір мови програмування — С або C++ для даного проекту.

На обраній мові програмування буде писатися програма, що реалізує відповідні алгоритми. Враховуючи, що в рамках вивчення дисципліни «Інформатика. Частина 1. Основи програмування та алгоритми» передбачається вивчення мови програмування С та виконання завдань на цій мові програмування, тому необхідно лівою кнопкою миші обрати мову програмування С, і після цього натиснути кнопку Next.

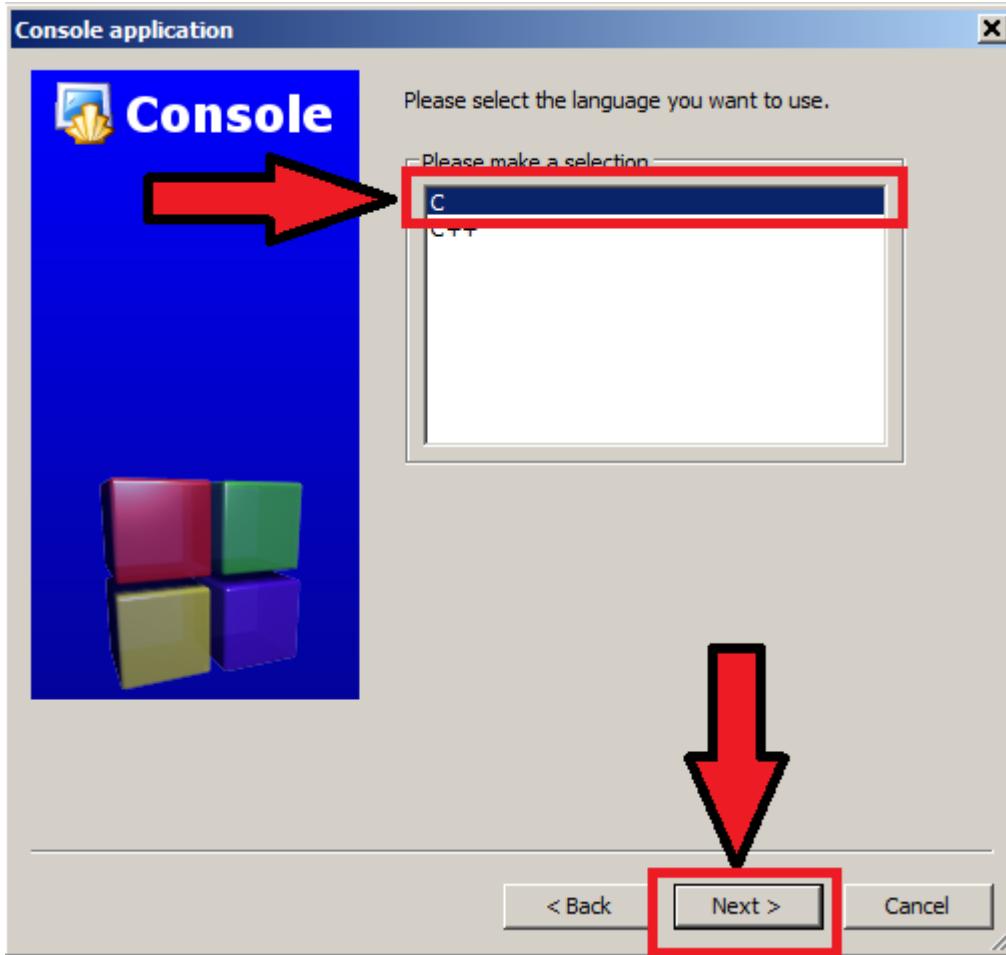


Рис. 1.9 — Вибір мови програмування С для проекту, що створюється

Наступне діалогове вікно, яке виникає і яке зображено на рис. 1.10, пропонує задати назву для нового проекту, що створюється, а також вказати місце на диску, де проект буде зберігатися. Важливо, щоб при виборі папки на диску для зберігання файлу проекту, у облікового запису користувача в операційній системі були встановлені права, що дозволяють запис файлів на обраний диск або в обраний каталог. В іншому випадку, можуть виникнути помилки на етапі зберігання змін, що були внесені в файл проекту, при роботі над проектом при написанні програми.

Після того, як були вказані ім'я проекту та каталог, в якому проект буде зберігатися, необхідно натиснути на кнопку Next, щоб перейти до діалогового вікна вибору компілятора. Приклад такого діалогового вікна показано на рис. 1.11.

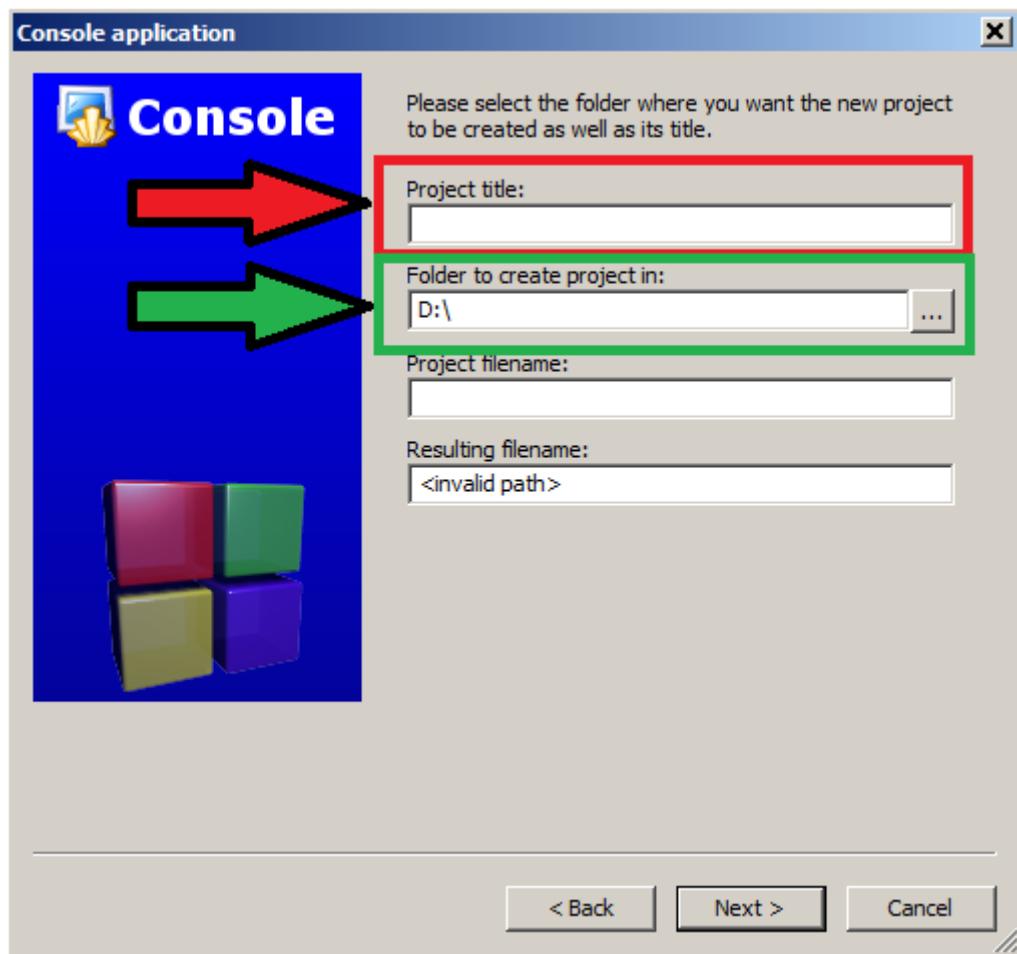


Рис. 1.10 — Введення назви проекту та вибір розміщення файлу проекту

По замовчуванню, в діалогову вікні вибору компілятора на рис. 1.11, у рядку Compiler може відображатися назва компілятора, який був встановлений при інсталяції на персональний комп’ютер інтегрованого середовища розробки Code::Blocks, — це компілятор GNU GCC.

Натисканням кнопки Finish завершуємо створення нового проекту і переходимо до вікна Code::Blocks, приклад якого показано на рис. 1.12.

У вікні Code::Blocks зліва буде відображені панель Management, де на вкладці Projects буде міститися піктограма із назвою проекту, яка була введена користувачем на етапі створення проекту в діалоговому вікні на рис. 1.10. Натиснувши на знак “+” біля паки Sources можна буде відобразити файли

проекту. По замовчуванню, буде відображатися файл main.c. Необхідно двічі курсором миші клікнути на піктограмі білого аркушу або на імені файлу main.c, щоб відкрити файл main.c, в якому буде виконуватися написання програми на мові C.

Вигляд вікна Code::Blocks, а також вміст файлу main.c показано на рис. 1.13.

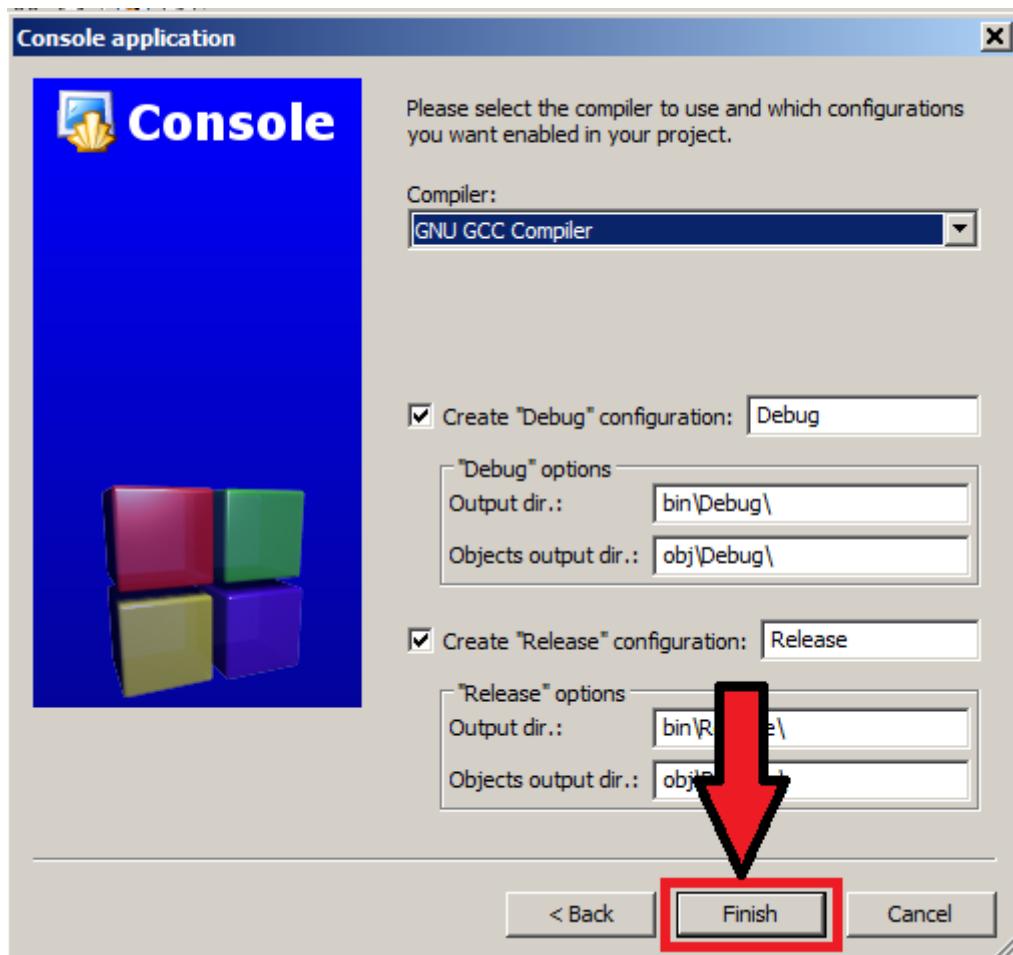


Рис. 1.11 — Діалогове вікно вибору компілятора

При створенні нового проекту Console application, створюється файл проекту, в якому міститься елементарна програма по виведенню на екран монітору в консольне вікно операційної системи повідомлення Hello World!, крім того, одразу в файлі main.c присутні такі директиви препроцесора:

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

Підключення заголовочних файлів стандартної бібліотеки `stdio.h` та `stdlib.h` дозволить викликати окремі стандартні бібліотечні функції, зокрема, підключення файлу `stdio.h` забезпечує можливість використовувати в програмі функцію `printf()` для виведення інформації на екран в командне вікно, а також функцію `scanf()` для зчитування даних з клавіатури.

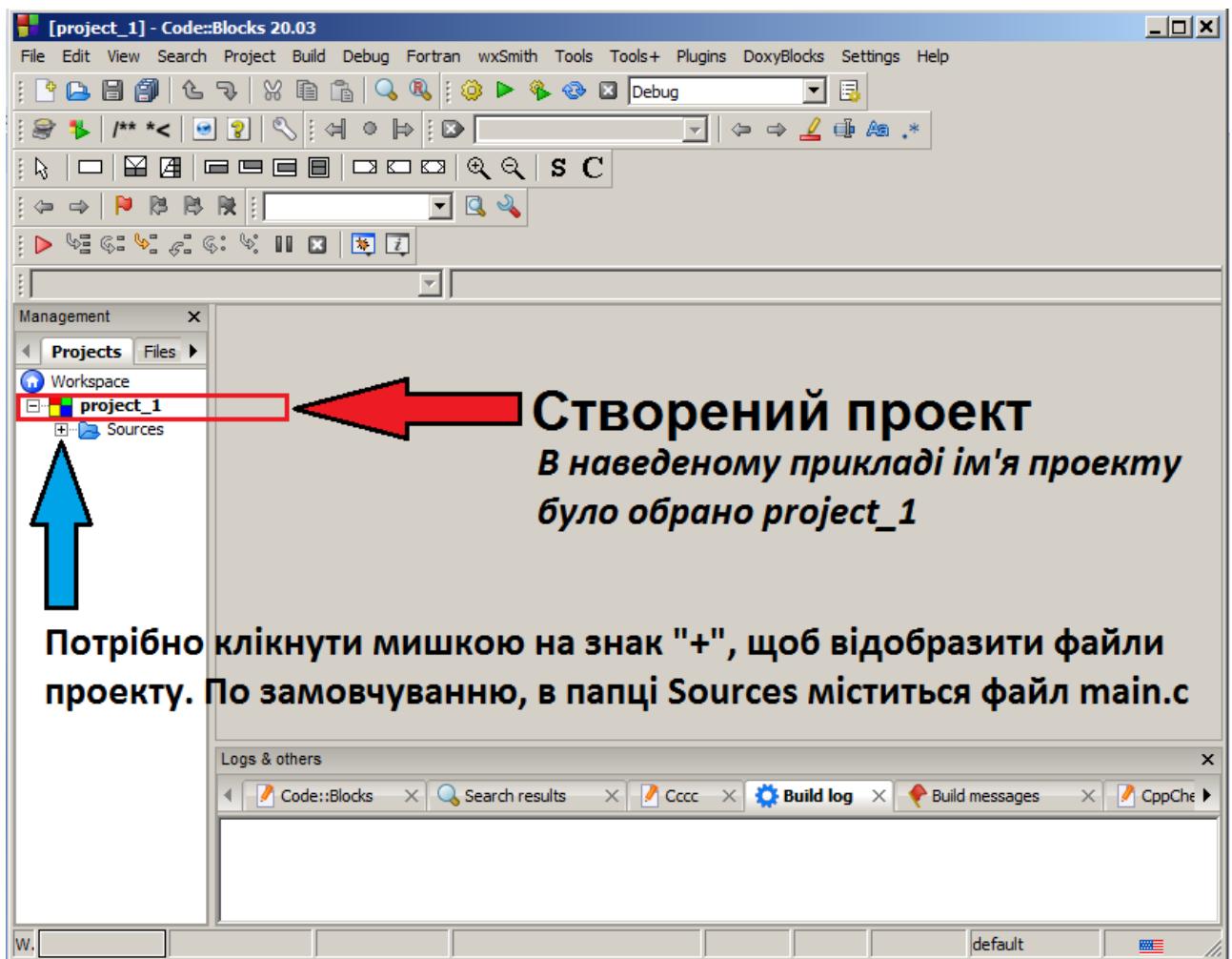


Рис. 1.12 — Вікно інтегрованого середовища розробки Code::Blocks

В разі необхідності використовувати інші функції стандартної бібліотеки, наприклад, математичні функції, або функції по роботі із символьними рядками тощо, — в такому разі потрібно підключати за допомогою директиви препроцесора `#include` відповідний заголовочний файл.

В файлі main.c проекту, що був створений, необхідно буде писати програму, яка буде реалізовувати відповідні алгоритми. Перед тим як записувати код певного алгоритму, який необхідно реалізувати, можна перевірити, чи буде проект, який створений по замовчуванню, в якому виводиться на екран лише повідомлення Hello World!, правильно компілюватися. Для цього, перед тим, як вносити будь-які зміни в файл main.c, можна натиснути клавішу **F9** або натиснувши на піктограму, яка виділена на рис.1.13, і перевірити правильність компіляції проекту. Якщо в цьому випадку ніяких помилок не виникло і на екрані в консольному вікні було відображене повідомлення Hello World!, можна приступати до внесення змін в файл main.c і записувати в нього свій програмний код.

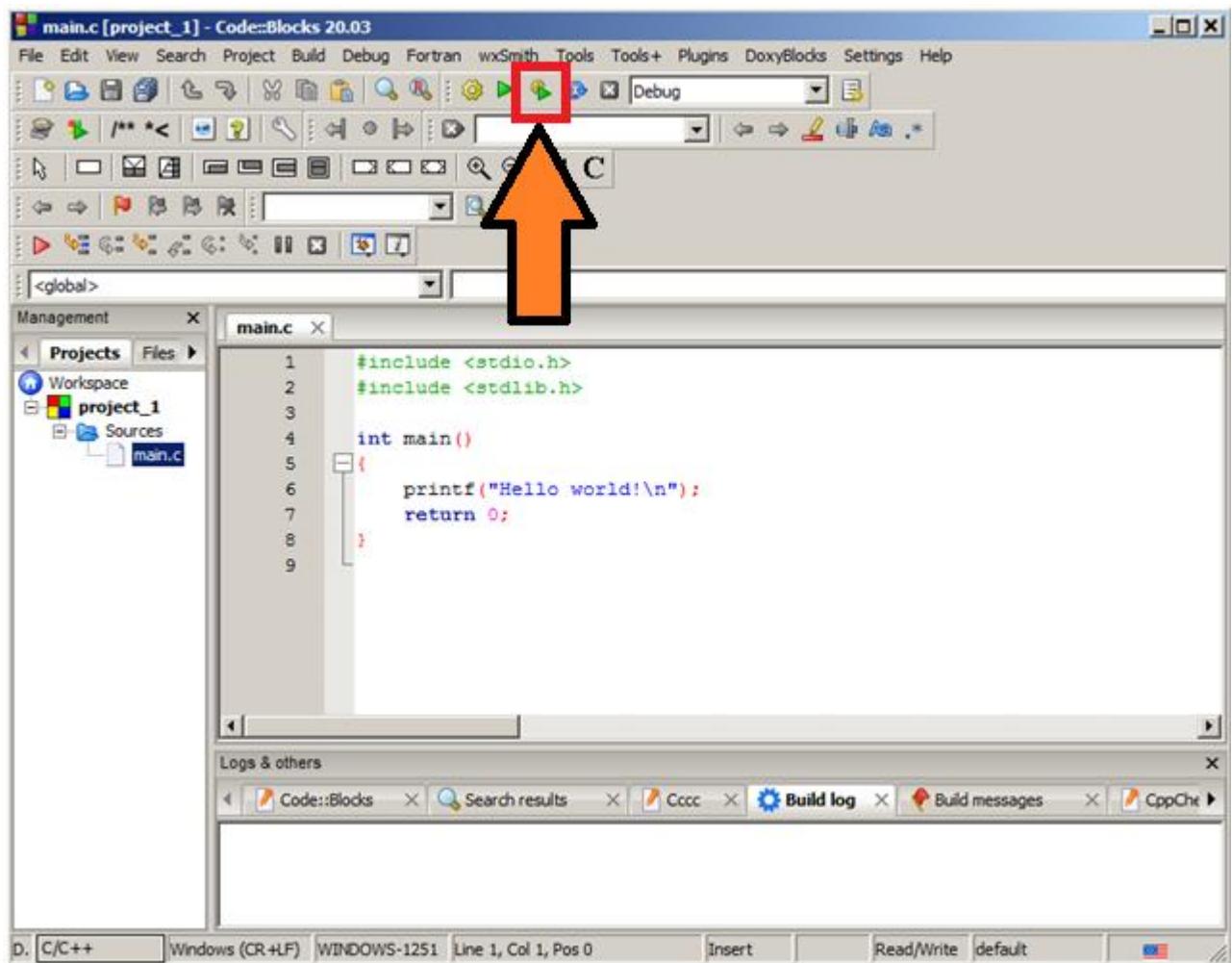


Рис.1. 13 — Файл main.c, що створюється по замовчуванню при створенні нового проекту Console application

2. СИСТЕМИ ЧИСЛЕННЯ

Дані, з якими доводиться мати справу, які потрібно обробляти, застосовуючи відповідні методи та алгоритми, найчастіше представлені у вигляді послідовності чисел. Навіть та інформація, яка на перший погляд, не має ніякого відношення до числових величин, наприклад, текстова інформація, що зберігається у файлі на комп’ютері, також певним чином для свого зберігання передбачає використання числових даних [1—3].

Найчастіше в повсякденному житті доводиться мати справу із десятковими числами. Десяткове число, наприклад, 5392, складається із окремих символів, кожен із яких розміщується на своїй позиції. Такі символи, які формують число, називаються цифрами. Цифри — це такі собі будівельні блоки, за допомогою яких можна утворити потрібне число, поставивши цифри на відповідні позиції.

В десятковій системі числення для утворення чисел використовуються цифри:

0
1
2
3
4
5
6
7
8
9

Таким чином, в загальному випадку десяткове число може бути представлено:

$$X = x_{n-1}x_{n-2}x_{n-3}\dots x_2x_1x_0,$$

де X — це десяткове число, яке складається із окремих цифр, що позначені $x_{n-1}, x_{n-2}, \dots, x_0$, де нижнім індексом показано номер позиції цифри в утвореному ними числі. Позиція цифри в числі називається *роздрядом*. Також *роздрядом* ще можуть називати значення цифри, що розміщується на відповідній позиції в числі. Таким чином, повертаючись до загального запису числа X можна говорити, що дане число складається із n роздрядів (номери роздрядів — від 0 до $n-1$). Кількість роздрядів в числі визначає розрядність числа.

Наприклад, число 5392 — це чотирьох розрядне число (кількість розрядів $n=4$), яке складається із розрядів: $x_0 = 2, x_1 = 9, x_2 = 3, x_3 = 5$.

В представленому прикладі $x_0 = 2$ — це наймолодший розряд, $x_3 = 5$ — це найстарший розряд.

Число, яке записане у відповідній системі числення, формується за допомогою відповідних символів (в наведеному прикладі ці символи представляють собою десяткові цифри). Такі символи ще називають алфавітом. Кількість символів в алфавіті визначає основу системи числення. Наприклад, в десятковій системі числення алфавіт складається із 10 символів — це 10 цифр — 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, — таким чином основа такої системи числення — 10. Основа системи числення буде позначатися символом q . Для десяткової системи числення $q=10$.

Для того щоб при записі числа відображати інформацію щодо системи числення, в якій це число записано, буде використовуватися нижній індекс із значенням основи відповідної системи числення. Наприклад, число 5392 в десятковій системі числення може бути записано такими способами:

$$5392_{10} \text{ або } (5392)_{10}$$

Десяткова система числення відноситься до так званої позиційної системи числення тому що значення кожної цифри в записі десяткового числа залежить від позиції цифри, тобто позиція цифри визначає вагу цієї цифри при формуванні значення числа. Виходячи із цього число 5392_{10} може бути представлено як суму добутків, де кожен добуток обраховується як результат

множення значення відповідної цифри на відповідний ваговий коефіцієнт, де значення вагового коефіцієнта визначається номером розряду цифри в числі. Враховуючи раніше наведену схему представлення нумерації розрядів, число 5392_{10} може бути записане таким чином:

$$5392_{10} = 5 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$$

Або в загальному випадку:

$$X = x_{n-1}x_{n-2}x_{n-3}\dots x_2x_1x_0 = x_{n-1}q^{n-1} + x_{n-2}q^{n-2} + \dots + x_1q^1 + x_0q^0,$$

де X — n -роздрядне десяткове число; x_i , $i = \overline{0, n-1}$ — десяткові розряди; $q=10$.

Діапазон можливих значень n -роздрядного десяткового числа становить $0\dots(10^n - 1)$. Наприклад, для 4-х розрядного числа — мінімальне значення становить 0, а максимальне значення буде визначатися як $10^4 - 1 = 10000 - 1 = 9999$.

Десяткова система числення знайшла широке застосування в повсякденному житті при виконанні простих арифметичних операцій в побуті, а також в науці при виконанні розрахунків в математиці, фізиці тощо. Однак, незважаючи на широке розповсюдження десяткової системи числення, в цифровій техніці та комп’ютерній техніці використовується інша система числення для зберігання інформації [1—3].

В комп’ютерній техніці для зберігання числових даних використовується двійкова система числення. Значення, яке записано в двійковій системі числення, ще називають двійковим кодом. Для кращого розуміння яким чином в пам’яті комп’ютера зберігаються дані в двійковому коді необхідно розібратися із особливостями двійкової системи числення та опанувати методику перетворення числових значень із десяткової системи числення в двійкову систему числення, а також навчитися виконувати зворотні перетворення — переводити числові значення із двійкової системи числення в десяткову систему числення.

2.1 Двійкова система числення

Двійкова система числення відноситься до позиційних систем числення. Тобто, аналогічно десятковій системі числення, вага двійкової цифри визначається позицією цієї цифри.

Двійкове число (або іншими словами — двійковий код) утворюється за допомогою алфавіту, що складається із двох символів — 0 (нуль) та 1 (одиниця). Таким чином, в двійковій системі числення двійковий код формується тільки із цих двох двійкових цифр. Основа двійкової системи числення $q=2$.

Аналогічно прийнятому раніше правилу — для вказування системи числення, в якій записана відповідна кодова послідовність, буде використовуватися нижній індекс із значенням основи відповідної системи числення. Приклад запису двійкового коду:

$$1011_2 \text{ або } (1011)_2$$

Двійковий нуль та двійкову одиницю ще називають бітом (одиночний біт). Комбінація із декількох бітів формує відповідне двійкове число (двійковий код).

Якщо в десятковій системі числення вага кожного розряду збільшується порівняно із попереднім розрядом в 10 разів, то в двійковій системі числення вага кожного розряду збільшується в 2 рази порівняно із вагою попереднього розряду [1, 2].

Наприклад, в приведеному двійковому коді 1011_2 — наймолодший розряд (крайній правий розряд) має вагу 1, а найстарший розряд (крайній лівий розряд) має вагу 8. Таким чином, в представленому варіанті ваги розрядів будуть такими: 1, 2, 4, 8.

Розглянемо два способи, які можуть використовуватися для перетворення десяткових чисел в двійковий код.

Спосіб 1. Перший спосіб передбачає послідовне виконання операції ділення, яка застосовується до десяткового числа (числа в десятковій системі числення), та зберігання остач від ділення, які визначаються на кожному кроці застосування операції ділення [2].

При перетворенні числа із *десяткової* системи числення в *двійкову* систему числення необхідно виконувати *ділення* на значення *основи системи числення* до якої перетворюється вхідне десяткове число.

Розглянемо на прикладі процедуру перетворення числа із десяткової системи числення в двійковий код.

Нехай обрано ціле десяткове число: 207_{10} .

Застосуємо до цього числа операцію ділення на основу систему числення до якої відбувається перетворення. Враховуючи, що відбувається перетворення до двійкової системи числення, то дільник буде дорівнювати 2. Ділення можна виконувати в стовпчик (рис. 2.1).

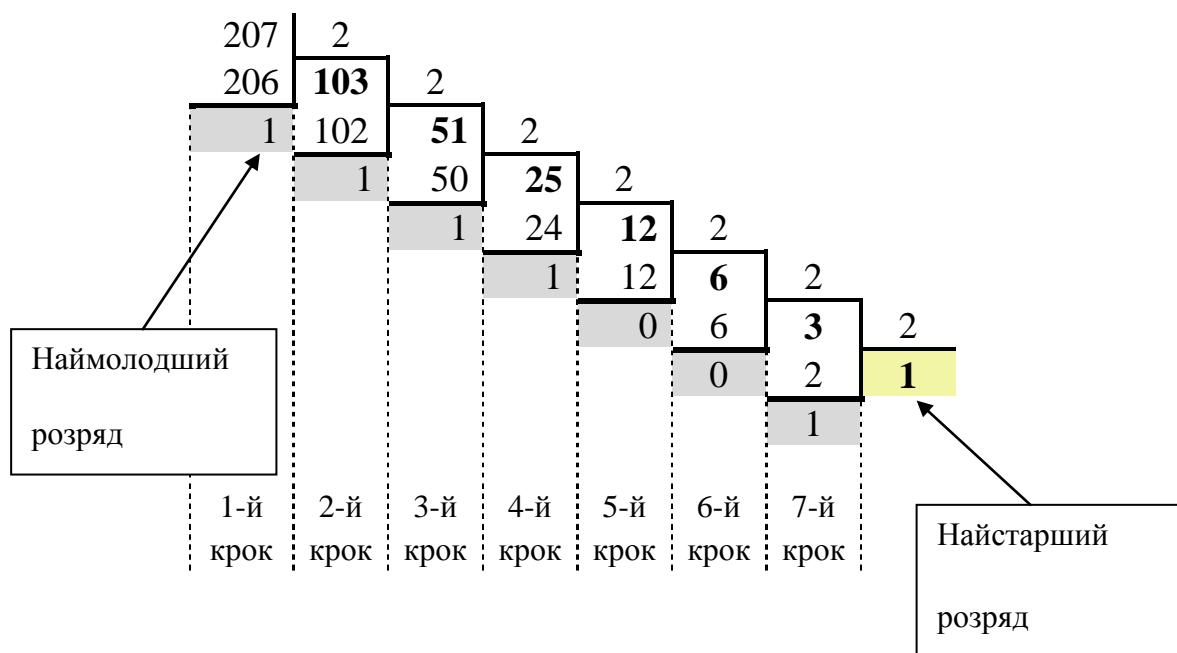


Рис. 2.1 — Перетворення десяткового числа із десяткової системи числення в двійкову систему числення

На *першому* кроці вхідне десяткове число ділиться на 2, окремо запишемо цю операцію:

$$\begin{array}{r} 207 \\ 206 \end{array} \overline{)103} \quad \begin{array}{r} 2 \\ 1 \end{array}$$

або можна записати таким чином:

$$\frac{207}{2} = 103\frac{1}{2}$$

Отже, в результаті ділення, отримаємо 103 цілих, і одиницю в остачі. Остачу зберігаємо. Надалі знайдена остача буде використовуватися для запису двійкового коду числа. Значення остачі, яка знайдена на першому кроці, буде виступати в якості наймолодшого розряду в двійковому коді. Наймолодший розряд записується в крайній правій позиції. Наймолодший розряд буде стояти в двійковому коді на позиції номер 0.

В наведеній схемі, ділене — це 207_{10} , дільник — це 2_{10} (основа системи числення до якої відбувається перетворення), частка — 103_{10} , остача — 1.

Якщо частка більша ніж 1, то потрібно продовжити процес ділення на основу системи числення (продовжити ділення на 2).

На *другому* кроці значення частки з попереднього кроку ділиться на 2, окремо запишемо цю операцію:

$$\begin{array}{r} 103 \\ 102 \end{array} \overline{)51} \quad \begin{array}{r} 2 \\ 1 \end{array}$$

або можна записати таким чином:

$$\frac{103}{2} = 51\frac{1}{2}$$

В результаті ділення остача дорівнює 1. Це значення необхідно зберегти. Воно буде виступати двійковим розрядом, який буде стояти на позиції 1 в двійковому коді. Якщо частка більша ніж 1, то процес повторюється. В даному випадку частка дорівнює 51_{10} . Тому відбувається перехід на третій крок.

На третьому кроці частка з попереднього кроку ділиться на 2. Запишемо цю операцію:

$$\begin{array}{r} 51 \\ 50 \\ \hline 1 \end{array} \quad \begin{array}{r} 2 \\ \hline 25 \end{array}$$

або можна записати таким чином:

$$\frac{51}{2} = 25\frac{1}{2}$$

В результаті ділення маємо: остача дорівнює 1 — це значення двійкового розряду, що буде стояти на позиції 2.

Частка дорівнює 25_{10} . Оскільки $25 > 1$, то процес ділення повторюється і відбувається перехід на четвертий крок.

На четвертому кроці частка з попереднього кроку ділиться на 2. Запишемо окремо цю операцію:

$$\begin{array}{r} 25 \\ 24 \\ \hline 1 \end{array} \quad \begin{array}{r} 2 \\ \hline 12 \end{array}$$

або можна записати таким чином:

$$\frac{25}{2} = 12\frac{1}{2}$$

Остача дорівнює 1 — цей біт буде стояти на позиції номер 3 в двійковій кодовій послідовності. Частка дорівнює 12_{10} , оскільки частка більше ніж 1, то відбувається перехід на наступний крок.

На п'ятому кроці повторюються операції, які аналогічні тим, що відбувалися на попередніх кроках:

$$\begin{array}{r} 12 \\ 12 \\ \hline 0 \end{array} \quad \begin{array}{r} 2 \\ \hline 6 \end{array}$$

або можна записати таким чином:

$$\frac{12}{2} = 6 \frac{0}{2}$$

Остача дорівнює 0 — це двійковій розряд, який буде розміщуватися на позиції 4. Частка дорівнює 6_{10} . Оскільки частка > 1 , переходимо до шостого кроку.

На шостому кроці відбуваються такі дії:

$$\begin{array}{r} 6 \\ 6 \end{array} \overline{)2} \quad \begin{array}{r} 6 \\ 6 \end{array} \overline{)3} \quad \underline{0}$$

або можна записати таким чином:

$$\frac{6}{2} = 3 \frac{0}{2}$$

Остача дорівнює 0 — це значення буде розміщуватися на позиції номер 5 в двійковому коді. Частка дорівнює 3_{10} . Оскільки $3 > 1$, то процес ділення продовжується:

На сьомому кроці:

$$\begin{array}{r} 3 \\ 2 \end{array} \overline{)2} \quad \begin{array}{r} 3 \\ 2 \end{array} \overline{)1} \quad \underline{1}$$

або запишемо таким чином:

$$\frac{3}{2} = 1 \frac{1}{2}$$

Остача дорівнює 1 — цей біт буде розміщуватися на позиції номер 6.

Частка дорівнює 1. Значення частки більше не може бути розділене на 2, тому процес ділення завершується. Значення частки, що дорівнює 1, буде найстаршим розрядом в двійкову коді, і це значення буде розміщуватися в розряді із номером позиції 7.

В схемі послідовного ділення на 2 в стовпчик, яка показана на рис. 2.1, значення остач, які отримані на кожному кроці, виділені сірим кольором.

Значення часток, які отримані на кожному кроці виділені жирним шрифтом. Остання знайдена частка (дорівнює 1), яка виступає в якості найстаршого розряду, виділена жовтим кольором.

Після процедури ділення залишається правильно записати результат. Наймолодший розряд записується в крайній правій позиції (номер цієї позиції 0). Найстарший розряд записується в крайній лівій позиції (в даному прикладі номер найстаршого розряду дорівнює 7). Всього розрядів 8. На рис. 2.2 показано отриманий двійковий код. Номер позиції розряду записано над значенням відповідного двійкового розряду:

<i>Номер позиції розряду:</i>	7	6	5	4	3	2	1	0
Двійковий код:	1	1	0	0	1	1	1	1

Рис. 2.2 — Розміщення найстаршого та наймолодшого розрядів в двійковому коді

Таким чином, десятковому числу 207_{10} відповідає двійковий код 11001111_2 .

Дуже часто виникає необхідність виконувати зворотне перетворення — перетворювати значення, що записано в двійковому коді, в десяткове число.

Якщо при перетворенні із десяткової системи числення в двійкову систему числення необхідно виконувати ділення на 2 (на основну систему числення), то при зворотному перетворенні — необхідно виконувати множення на 2 у відповідній степені [2].

Процедура зворотного перетворення числового значення із двійкового коду в десяткову систему числення наступна: необхідно кожен двійковий розряд помножити на ваговий коефіцієнт, який визначається як 2, що

підноситься у відповідну степінь, де значення степені визначається порядковим номером двійкового розряду. Нумерація розрядів починається з 0. Якщо двійковий код складається із n розрядів, то номер наймолодшого розряду — це 0, і відповідно значення наймолодшого двійкового розряду множиться на 2^0 , більш старший розряд (який знаходиться на позиції 1) множиться на 2^1 і т.д., найстарший розряд (який знаходиться на позиції $n-1$) множиться на 2^{n-1} . Отримані результати додаються [2].

Для прикладу двійкового коду, який показаний на рис. 2.2, значення десяткового числа, яке отримується шляхом перетворення двійкової кодової послідовності із двійкової системи числення в десяткову, буде визначатися таким чином:

$$1 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 1 \times 2^6 + 1 \times 2^7 = \\ = 1 + 2 + 4 + 8 + 64 + 128 = 207_{10}$$

В загальному випадку, якщо задано двійковий код (двійкове число) A_2 що складається із n розрядів (значення кожного двійкового розряду позначено як a_i , $i = \overline{0, n-1}$):

$$A_2 = a_{n-1}a_{n-2}...a_2a_1a_0,$$

то щоб виконати перетворення двійкової послідовності в десяткову систему числення і знайти відповідне десяткове значення, яке позначимо як X_{10} , необхідно виконати наступні дії:

$$X_{10} = a_{n-1}q^{n-1} + a_{n-2}q^{n-2} + \dots + a_2q^2 + a_1q^1 + a_0q^0,$$

де $q = 2$.

Розрядність двійкового коду визначає діапазон можливих значень, в якому можуть знаходитися відповідні їм десяткові значення.

Наприклад, однорозрядна двійкова кодова послідовність складається лише із одного розряду (із одного біту). Значення, які може приймати біт — 0 або 1. В десятковій системі числення двійковому 0 буде відповідати десятковий 0:

$$0_2 = 0 \times 2^0 = 0 \times 1 = 0_{10}$$

Значенню 1_2 буде відповідати в десятковій системі числення значення 1_{10} :

$$1_2 = 1 \times 2^0 = 1 \times 1 = 1_{10}$$

Таким чином, однорозрядне двійкове число при перетворенні в десяткову систему числення може дорівнювати 0_{10} або 1_{10} . Дворозрядне двійкове число складається із двох бітів. Мінімальне значення, яке в цьому випадку може бути записане — це послідовність із двох двійкових нулів: 00_2 . В десятковій системі числення цій кодовій послідовності відповідає значення 0_{10} :

$$00_2 = 0 \times 2^0 + 0 \times 2^1 = 0 \times 1 + 0 \times 2 = 0 + 0 = 0_{10}$$

Максимальне значення, яке може бути збережене в дворозрядному двійковому коді — це двійковий код 11_2 . В десятковій системі числення цій кодовій послідовності відповідає значення 3_{10} :

$$11_2 = 1 \times 2^0 + 1 \times 2^1 = 1 \times 1 + 1 \times 2 = 1 + 2 = 3_{10}$$

Таким чином, дворозрядний двійковий код може зберігати кодові комбінації, які будуть відповідати наступним десятковим числам: 0_{10} , 1_{10} , 2_{10} , 3_{10} .

В загальному випадку, n -розрядне двійкове число (двійковий код) може відповідати цілому десятковому значенню із діапазону:

$$0_{10} \dots (2^n - 1)_{10}$$

Наприклад, восьмирозрядне ($n=8$) двійкове число, при перетворенні значення із двійкової системи числення в десяткову систему числення, може набувати цілого значення із діапазону:

$$0_{10} \dots (2^8 - 1)_{10} \text{ тобто } 0_{10} \dots 255_{10}$$

В цьому випадку значенню 0_{10} буде відповідати восьмирозрядний двійковий код 00000000_2 . Значення 255_{10} буде відповідати восьмирозрядний двійковий код 11111111_2 .

Розуміння того яким чином співвідносяться значення в десятковій системі числення і відповідні їх двійкові коди є дуже важливим, зокрема і при виборі типів даних для змінних та констант на етапі написання програм на мові програмування C.

Другий спосіб, який може бути застосований для перетворення цілих чисел із десяткової системи числення в двійкову систему числення, базується на тому, що кожне десяткове число може бути представлене як сума деяких доданків, де відповідні доданки визначаються як двійка, що піднесена до деякої степені. Значення степені відповідає номеру двійкового розряду. Визначення значень номерів двійкових розрядів (або іншими словами — визначення значень степенів), які приймають участь у формуванні доданків, які в сумі формують значення десяткового числа, дозволяє виконати перетворення числа із десяткової системи числення в двійкову систему числення.

Спосіб 2. Цей спосіб передбачає побудову таблиці, в яку треба записати номери двійкових розрядів, а також для зручності перетворення — внесемо в таблицю значення двійки у відповідній степені (степінь визначається номером відповідного розряду). Кількість стовпців в таблиці (кількість степенів двійки) визначається розрядністю двійкового коду, який в свою чергу залежить від максимального значення десяткового числа, яке може приймати участь в перетворенні в двійкову систему числення. Для приведеного далі прикладу покладається, що десяткове значення, яке буде перетворюватися в двійкову систему числення, може знаходитись в діапазоні $0_{10} \dots 255_{10}$. Максимальне значення 255_{10} для свого збереження в двійковому коді потребує вісім двійкових розрядів. Тому побудуємо таблицю, яка буде містити значення двійки у відповідній степені, де величина степені буде набувати значення від 0 до 7 (табл. 2.1) [4].

Табл 2.1 — Значення степенів числа 2

Значення, що виступають в якості доданків	128	64	32	16	8	4	2	1
Номер розряду	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Двійковий код	7	6	5	4	3	2	1	0

Мінімальне значення, яке може бути збережене в восьмирозрядному двійковому коді — 00000000_2 , що відповідає значенню 0_{10} .

Максимальне значення, яке може бути збережене в восьмирозрядному двійковому коді — 11111111_2 , що відповідає значенню 255_{10} .

В табл. 2.1 в верхньому рядку жирним шрифтом записані значення степені двійки, — ці величини виступають в якості доданків за допомогою яких можна сформувати будь-яке число із діапазону $0_{10} \dots 255_{10}$.

При виконанні перетворення числа із десяткової системи числення в двійковий код необхідно визначити які саме доданки із представлених в верхньому рядку таблиці, в сумі дають десяткове число, яке перетворюється в двійкову систему числення. Для тих доданків, які приймають участь у формуванні десяткового числа, у відповідний рядок, який називається «Двійковий код» ставиться 1. Якщо відповідний доданок не приймає участь у формуванні числа — ставиться 0.

Наприклад, для числа 0_{10} жоден із доданків не використовується, тому для запису восьмирозрядної двійкової кодової послідовності необхідно нижній рядок таблиці заповнити нулями (табл. 2.2)

Табл 2.2 — Двійковий код числа нуль

<i>Значення, що виступають в якості доданків</i>	128	64	32	16	8	4	2	1
<i>Номер розряду</i>	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
<i>Двійковий код</i>	0	0	0	0	0	0	0	0

Наприклад, для числа 255_{10} всі доданки із верхнього рядка таблиці використовуються для формування даного значення, тому двійковий код для даного десяткового числа буде мати вигляд як показано в табл. 2.3.

Табл 2.3 — Двійковий код числа 255_{10}

<i>Значення, що виступають в якості доданків</i>	128	64	32	16	8	4	2	1
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
<i>Номер розряду</i>	7	6	5	4	3	2	1	0
<i>Двійковий код</i>	1	1	1	1	1	1	1	1

В якості прикладу, виконаємо перетворення десяткового числа 207_{10} в двійкову систему числення.

В процесі перетворення числа 207_{10} в двійковий код треба визначити які саме із доданків: 128, 64, 32, 16, 8, 4, 2, 1 будуть в сумі дорівнювати значенню 207_{10} . Процедуру по виконанню перетворення можна представити у вигляді послідовності кроків, на кожному кроці виконуємо аналіз поточного значення та відповідного доданку із таблиці. Якщо доданок входить в суму, що формує число — ставимо у відповідний стовпчик 1, віднімаємо цей доданок від числа, яке перетворюється у двійкову систему числення, і переходимо на наступний крок. Якщо відповідний доданок не використовується у формуванні десяткового числа — ставимо у відповідний стовпчик 0, і переходимо на наступний крок. Починати описаний процес можна з будь-якого кінця таблиці. Наприклад, почнемо із аналізу доданку, який відповідає найстаршому двійковому розряду — це число 128:

- 1) Порівнюємо число 207_{10} (число яке в перетворюється в двійкову систему числення) і число 128_{10} . Якщо десяткове число більше або рівне значенню 128_{10} то найстаршому двійковому розряду (позначимо цей розряд a_7) присвоюється значення 1 (отже, $a_7 = 1$). Із десяткового числа віднімається значення 128_{10} і якщо отримане значення більше 0, тоді переходимо на наступний крок, на якому аналізуємо наступний доданок із таблиці. В приведеному прикладі:

$$207_{10} - 128_{10} = 79_{10}$$

2) Порівнюємо результат, що отриманий на попередньому кроці (число 79_{10}) і наступний можливий доданок із таблиці — це число 64_{10} . В даному випадку, $79_{10} \geq 64_{10}$ — отже 64_{10} виступає в якості доданка суми, яка формує значення 79_{10} . Відповідному двійковому розряду присвоюється значення 1 (розряд $a_6 = 1$), і віднімаємо число 64_{10} від числа 79_{10} :

$$79_{10} - 64_{10} = 15_{10}$$

3) Порівнюємо результат, який отримано на попередньому кроці (число 15_{10}) і відповідний можливий доданок із таблиці — це число 32_{10} . В даному випадку $15_{10} < 32_{10}$, отже значення 32_{10} не є доданком, тобто це значення не підходить, тому відповідному двійковому розряду присвоюємо значення 0 (розряд $a_5 = 0$). Значення числа, яке продовжує приймати участь в перетворенні, залишається рівним 15_{10} .

4) Порівнюємо результат із попереднього кроку (число 15_{10}) і відповідний доданок із таблиці — це число 16_{10} . В даному випадку $15_{10} < 16_{10}$, отже значення 16_{10} не є доданком, який використовується у формуванні поточного значення, тому відповідному двійковому розряду присвоюємо значення 0 (розряд $a_4 = 0$). Значення числа залишається рівним 15_{10} .

5) Порівнюємо результат із попереднього кроку (число 15_{10}) і відповідний доданок із таблиці — це число 8_{10} . В даному випадку $15_{10} \geq 8_{10}$, отже значення 8_{10} виступає в якості доданка, тому відповідному двійковому розряду в таблиці присвоюємо значення 1 (розряд $a_3 = 1$), і віднімаємо число 8_{10} від числа 15_{10} :

$$15_{10} - 8_{10} = 7_{10}$$

6) Порівнюємо результат із попереднього кроку (число 7_{10}) і відповідний доданок із таблиці — це число 4_{10} . В даному випадку $7_{10} \geq 4_{10}$, отже значення 4_{10} виступає в якості доданка, тому відповідному двійковому розряду в таблиці присвоюємо значення 1 (розряд $a_2 = 1$), і віднімаємо число 7_{10} від числа 4_{10} :

$$7_{10} - 4_{10} = 3_{10}$$

- 7) Порівнюємо результат із попереднього кроку (число 3_{10}) і відповідний доданок із таблиці — це число 2_{10} . В даному випадку $3_{10} \geq 2_{10}$, отже значення 2_{10} виступає в якості доданка, тому відповідному двійковому розряду в таблиці присвоюємо значення 1 (розряд $a_1 = 1$), і виконуємо віднімання числа 2_{10} від числа 3_{10} :

$$3_{10} - 2_{10} = 1_{10}$$

- 8) Порівнюємо результат із попереднього кроку (число 1_{10}) і відповідний доданок із таблиці — це число 1_{10} . В даному випадку $1_{10} \geq 1_{10}$, отже значення 1_{10} виступає в якості доданка, тому відповідному двійковому розряду в таблиці присвоюємо значення 1 (розряд $a_0 = 1$), і виконуємо віднімання числа 1_{10} від числа 1_{10} :

$$1_{10} - 1_{10} = 0_{10}$$

Виконавши останній крок, після віднімання 1_{10} , отримали значення 0_{10} . Таким чином, було виконано розкладання числа 207_{10} на такі доданки:

$$207_{10} = 128_{10} + 64_{10} + 8_{10} + 4_{10} + 2_{10} + 1_{10}$$

Відповідний двійковий код, який був отриманий в ході описаного способу перетворення десяткового значення в двійкову систему числення відображеній у табл. 2.4.

Табл 2.4 — Двійковий код числа 207_{10}

<i>Значення, що виступають в якості доданків</i>	128	64	32	16	8	4	2	1
<i>Номер розряду</i>	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
<i>Двійковий код</i>	1	1	0	0	1	1	1	1

Зворотне перетворення — із двійкової системи числення в десяткову систему числення, — також може бути виконане із використанням даної таблиці. При зворотному перетворенні, потрібно в нижній рядок таблиці на відповідні позиції вписати двійкові розряди. Біти, які дорівнюють 1,

визначають які саме доданки будуть в сумі давати шукане значення в десятковій системі числення [4].

Наприклад, нехай потрібно значення, що записано в двійковому коді — 10011100_2 , перетворити в десяткову систему числення, скориставшись таблицею. Для цього запишемо двійкові розряди на відповідні позиції в табл. 2.5.

Табл 2.5 — Перетворення коду 10011100_2 в десяткову систему числення

<i>Значення, що виступають в якості доданків</i>	128	64	32	16	8	4	2	1
<i>Номер розряду</i>	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
<i>Двійковий код</i>	1	0	0	1	1	1	0	0

В наведеному прикладі, біти, що дорівнюють 1, стоять у стовпцях із такими числами: 128_{10} , 16_{10} , 8_{10} , 4_{10} . Таким чином, шукане десяткове число X_{10} , якому відповідає двійковий код 10011100_2 , буде дорівнювати:

$$X_{10} = 128_{10} + 16_{10} + 8_{10} + 4_{10} = 156_{10}$$

У випадку необхідності операування значеннями із іншого діапазону ніж $0 \dots 255$, дана таблиця може бути адаптована під відповідний діапазон значень.

2.2 Вісімкова система числення

Крім двійкої системи числення, можуть використовуватися також інші системи числення.

Розглянемо яким чином виконується перетворення значень із *десяткової* системи числення у *вісімкову* систему числення [1—4].

Для формування вісімкових кодових послідовностей використовуються вісімкові цифри: 0, 1, 2, 3, 4, 5, 6, 7. *Основа вісімкової системи числення* $q=8$.

При перетворенні значення із десяткової системи числення у вісімкову систему числення необхідно послідовно виконувати операцію ділення на 8 по аналогії із перетворенням із десяткової системи числення в двійкову систему числення. В ході перетворення потрібно бути уважним і звертати увагу на значення вісімкових розрядів, які отримуються в ході перетворення. Значення вісімкових розрядів не можуть виходити за діапазон 0...7 [2, 4].

В якості прикладу виконаємо перетворення десяткового числа 207_{10} в вісімкову систему числення:

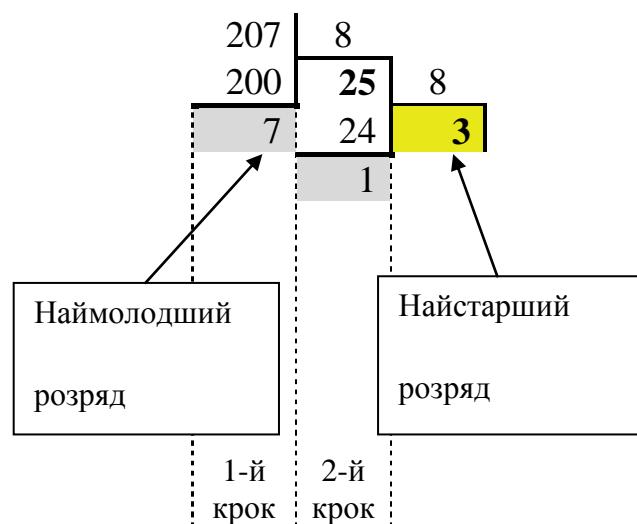


Рис. 2.3 — Перетворення десяткового числа із десяткової системи числення в вісімкову систему числення

На рис. 2.3 показаний механізм перетворення десяткового значення у вісімковий код. Ця процедура аналогічна тій, що була описана при розгляді питання перетворення десяткового числа у двійкову систему числення і відображена на рис. 2.1. При записі результату важливо в правильній послідовності записати вісімкові розряди — наймолодший розряд записується в крайньому правому положенні, а найстарший розряд — в крайньому лівому положенні. Для наведеного прикладу:

$$207_{10} = 317_8$$

В загальному випадку, вісімковий код, що складається із n розрядів, може бути представлений таким чином:

$$C_8 = c_{n-1}c_{n-2}\dots c_2c_1c_0$$

При зворотному перетворенні, — для того, щоб виконати перетворення із вісімкової системи числення в десяткову систему числення, необхідно кожен вісімковий розряд помножити на множник, який визначається як 8 у відповідній степені, де значення степені визначається номером відповідного вісімкового розряду, і отримані добутки додати. Знайдена сума і буде шуканим десятковим числом. Таким чином, в загальному випадку, маючи вісімковий код $C_8 = c_{n-1}c_{n-2}\dots c_2c_1c_0$, відповідне десяткове значення X_{10} буде визначатися так:

$$X_{10} = c_{n-1} \times 8^{n-1} + c_{n-2} \times 8^{n-2} + \dots + c_2 \times 8^2 + c_1 \times 8^1 + c_0 \times 8^0$$

При перетворенні вісімкового коду в десяткову систему числення можна сформувати схожу таблицю, яка застосовувалася для перетворення із двійкового коду в десяткову систему числення. Наприклад, для заданого трьохрозрядного вісімкового коду 317_8 заповнена таблиця буде мати такий вигляд як показано в табл. 2.6 [4].

Табл 2.6 — Перетворення вісімкового коду в десяткову систему числення

<i>Значення, що виступають в якості вагових коефіцієнтів розрядів</i>	64	8	1
	8^2	8^1	8^0
<i>Номер розряду</i>	2	1	0
Вісімковий код	3	1	7

Використовуючи табл. 2.6, виконаємо перетворення вісімкового коду 317_8 в десяткову систему числення — кожен вісімковий розряд множиться на відповідний ваговий коефіцієнт, і після цього знаходиться сума добутків, в результаті чого отримується шукане десяткове значення:

$$317_8 = 3 \times 8^2 + 1 \times 8^1 + 7 \times 8^0 = 3 \times 64 + 1 \times 8 + 7 \times 1 = 192 + 8 + 7 = 207_{10}$$

Якщо передбачається виконання перетворення вісімкових кодів, які складаються із більшої кількості розрядів ніж 3, то табл. 2.6 може бути відповідним чином змінена.

2.3 Шістнадцяткова система числення

В комп'ютерній техніці широко використовується шістнадцяткова система числення. Для формування шістнадцяткових кодів використовуються цифрові символи та буквенні символи. Нижче наведено шістнадцятові символи, де біля буквених символів вказано їх числове значення, яке записане у десятковій системі числення [1, 2, 4] :

0
1
2
3
4
5
6
7
8
9
A = 10 ₁₀
B = 11 ₁₀
C = 12 ₁₀
D = 13 ₁₀
E = 14 ₁₀
F = 15 ₁₀

Для того щоб виконати перетворення числового значення із десяткової системи числення в шістнадцяткову систему числення, необхідно виконувати операції ділення на основу системи числення, до якої відбувається

перетворення, тобто необхідно виконувати ділення на 16. Шістнадцятковий код отримується аналогічним чином як це було розглянуто при виконанні перетворень десяткового числа в двійкову систему числення або у вісімкову систему.

Розглянемо на прикладі яким чином отримати шістнадцятковий код для числа 207_{10} , застосовуючи для цього раніше описаний спосіб перетворення (рис. 2.4):

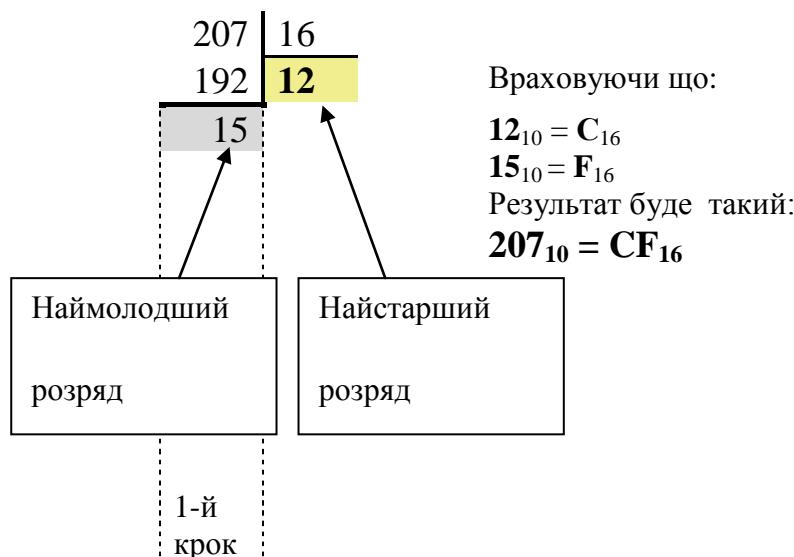


Рис. 2.4 — Перетворення числа із десяткової системи числення в шістнадцяткову систему числення

При записі остаточного результату важливо пам'ятати, що дворозрядні значення остач та частки, які отримуються в ході ділення десяткового числа на 16, необхідно замінити відповідними буквеними символами шістнадцяткової системи числення. Тому, при перетворенні числа 207_{10} в шістнадцяткову систему числення — результат буде CF_{16} , а не 1215_{16} !!!

В загальному випадку, шістнадцяткове число (шістнадцятковий код), який складається із n розрядів, можна записати так:

$$H_{16} = h_{n-1}h_{n-2}\dots h_2h_1h_0,$$

де h_i , $i = \overline{0, n-1}$ — шістнадцяткові розряди, які можуть набувати значення: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

При зворотному перетворенні — при перетворенні значення із шістнадцяткового коду в десяткову систему числення, — кожен шістнадцятковий розряд треба помножити на свій ваговий коефіцієнт, який визначається шляхом піднесення значення 16 у відповідну степінь, і знайти суму відповідних добутків. Наприклад, якщо заданий n -роздрядний шістнадцятковий код $H_{16} = h_{n-1}h_{n-2}\dots h_2h_1h_0$, то десяткове число X_{10} , яке буде отримуватися в ході перетворення шістнадцяткового коду в десяткову систему числення, буде визначатися таким чином:

$$X_{10} = h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

При виконанні перетворення значення із шістнадцяткової системи числення в десяткову систему числення можна скористатися таблицею, в якій будуть записувати відповідні вагові коефіцієнти для відповідного шістнадцяткового розряду. Розмір таблиці буде залежати від розрядності шістнадцяткових чисел, які потрібно перетворити в десяткову систему числення. Наприклад, якщо передбачається виконувати перетворення чотирьохроздяних шістнадцяткових кодів в десяткову систему числення, то для цього може бути використана табл. 2.7 [2, 4].

Табл 2.7 — Перетворення шістнадцяткового коду в десяткове число

<i>Значення, що виступають в якості вагових коефіцієнтів доданків</i>	4096	256	16	1
	16^3	16^2	16^1	16^0
<i>Номер розряду</i>	<i>3</i>	<i>2</i>	<i>1</i>	<i>0</i>
<i>Шістнадцятковий код</i>	A	2	F	E

Наприклад, для перетворення значення шістнадцяткового чотирьохроздядного коду A2FE₁₆ в десяткову систему числення, необхідно кожен шістнадцятковий розряд записати на свою позицію в таблиці,

помножити на відповідний ваговий коефіцієнт, і знайти суму. У випадку, коли шістнадцяткове число містить розряди, які позначені буквеними символами, необхідно в підрахунках використовувати відповідні їх числові значення. Для наведеного в табл. 2.7 прикладу, шукане значення X_{10} в десятковій системі числення буде визначатися таким чином:

$$\begin{aligned} X_{10} &= E \times 16^0 + F \times 16^1 + 2 \times 16^2 + A \times 16^3 = \\ &= 14 \times 16^0 + 15 \times 16^1 + 2 \times 16^2 + 10 \times 16^3 = \\ &= 14 \times 1 + 15 \times 16 + 2 \times 256 + 10 \times 4096 = \\ &= 14 + 240 + 512 + 40960 = 41726_{10} \end{aligned}$$

2.4 Методика швидкого перетворення двійкового коду у вісімковий код або шістнадцятковий код. Зворотне перетворення

Часто виникає необхідність виконувати перетворення значення, наприклад, із шістнадцяткової системи числення у двійкову систему числення, або навпаки. При цьому, раніше розглянуті варіанти перетворень дозволяли виконувати перетворення значень між такими системами числення:

1. *Десяткова* система числення \leftrightarrow *Двійкова* система числення
2. *Десяткова* система числення \leftrightarrow *Вісімкова* система числення
3. *Десяткова* система числення \leftrightarrow *Шістнадцяткова* система числення

Таким чином, якби потрібно було перетворити значення із *двійкової* системи числення в *шістнадцяткову* систему числення, то необхідно було б зробити такі кроки:

- 1) виконати перетворення значення із *двійкової* системи числення в десяткову систему числення;
- 2) виконати перетворення значення із *десяткової* системи числення, яке отримано на попередньому кроці, в *шістнадцяткову* систему числення.

Тобто, виходить, що перетворення в десяткову систему числення виступає в якості проміжного етапу. Це може бути певним недоліком такого підходу. Оскільки вимагає виконання додаткових затрат на перетворення із однієї системи числення в десяткову систему числення. А потім із десяткової системи числення необхідно виконати перетворення значення до потрібної системи числення. Методика швидкого перетворення дозволяє подолати цей недолік.

Методика швидкого перетворення може бути застосована для [2, 4]:

- перетворення значення із *двійкової* системи числення у *вісімкову* систему числення, і навпаки.
- перетворення значення із *двійкової* системи числення у *шістнадцяткову* систему числення, і навпаки.

При застосуванні методики швидкого перетворення між двійковою і вісімковою системами числення необхідно поставити у відповідність кожній вісімковій цифрі її двійковий код, причому відповідний двійковий код повинен складатися із трьох розрядів (табл. 2.8).

Табл 2.8 — Двійковий код вісімкових цифр

Вісімкова цифра	Трьохроздрядний двійковий код для відповідної вісімкової цифри
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

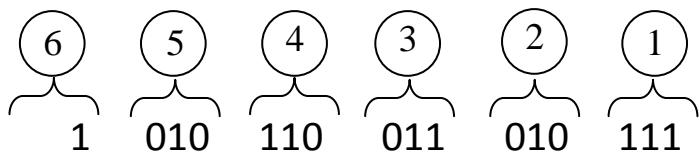
Розглянемо на прикладі методику швидкого перетворення значення із двійкового коду в вісімковий код.

Нехай задано наступний двійковий код:

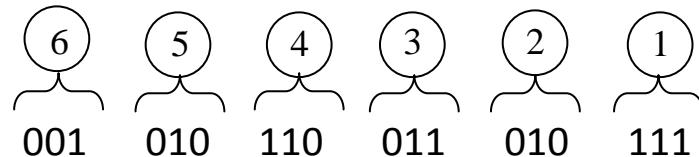
$$1010110011010111_2$$

Необхідно виконати його перетворення в вісімкову систему числення.

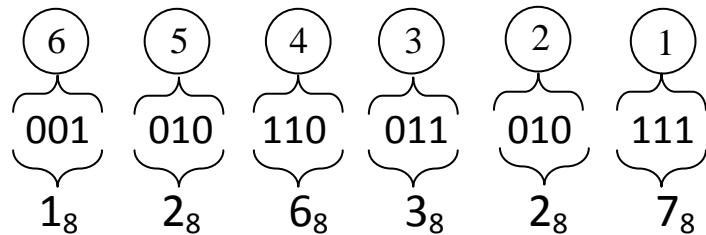
Для цього потрібно в двійковому коді виділи групи (набори) по три розряди (таку групу із трьох розрядів ще називають тріадою або триплетом), причому важливо відповідні групи по три розряди виділяти, починаючи з наймолодшого розряду (крайнього правого). Нижче показані відповідні тріади розрядів, зверху над кожною тріадою записано порядковий номер тріади, також для зручності візуального сприйняття між окремими тріадами зроблено відступи:



В тріаді № 6 відсутні два старші розряди. В разі необхідності — на позицію незаповнених старших розрядів, можна записати нулі, які ніяким чином не будуть впливати на величину числового значення:



Після цього, кожній тріаді ставиться у відповідність вісімкова цифра (трьохроздріні двійкові коди для відповідних вісімкових цифр показано в табл. 2.8). Запишемо це в такому вигляді:



Остаточний результат перетворення буде таким:

$$1010110011010111_2 = 126327_8$$

У випадку, коли необхідно виконати перетворення значення із вісімкової системи числення у двійкову систему числення, необхідно виконати зворотні дії до описаної вище процедури. Потрібно кожному вісімковому розряду поставити у відповідність його трьохроздядний двійковий код, і таким чином, отримати двійковий код для вхідного вісімкового коду. Розглянемо на прикладі цю процедуру. Нехай в двійкову систему числення необхідно перетворити наступний вісімковий код:

$$273540_8$$

Запишемо для кожного вісімкового розряду його відповідний трьохроздядний двійковий код із табл. 2.8 (для зручності візуального сприйняття, між окремими вісімковими розрядами вхідного вісімкового коду зроблено відступи):

2_8	7_8	3_8	5_8	4_8	0_8
{010}	{111}	{011}	{101}	{100}	{000}

Таким чином, отриманий двійковий код буде мати вигляд:

$$010111011101100000_2$$

В разі необхідності, старші нульові розряди, які не впливають на величину числа, можуть бути опущені (старші нульові розряди, які не змінюють величину значення називаються незначущими розрядами). Таким чином, остаточний результат може бути представлений так:

$$273540_8 = 10111011101100000_2$$

Схожим чином реалізується швидке перетворення значення із двійкової системи числення в шістнадцяткову систему числення, і навпаки.

Для цього необхідно сформувати таблицю відповідності кожного шістнадцяткового символу та його двійкового коду. На відміну від вісімкових цифр, кожному шістнадцятковому символу повинен відповідати чотирьохроздядний двійковий код (табл. 2.9).

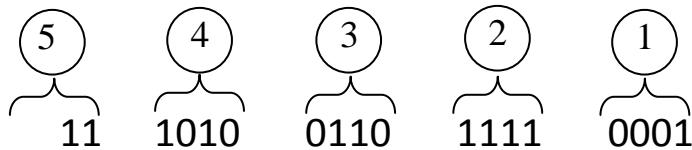
Табл 2.9 — Двійковий код шістнадцяткових цифр

Шістнадцятковий символ	Чотирьохроздядний двійковий код для відповідного шістнадцяткового символу
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

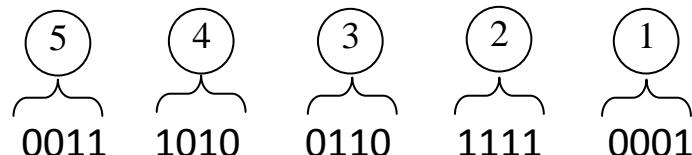
Розглянемо на прикладі застосування методики швидкого перетворення. Нехай в шістнадцяткову систему числення необхідно перетворити такий двійковий код:

$$111010011011110001_2$$

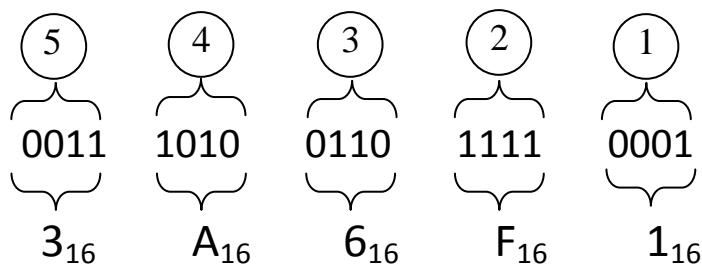
Необхідно виділити групи (набори) по чотири розряди, починаючи із наймолодшого розряду (група із чотирьох розрядів називається тетрадою). Нижче показані відповідні тетради розрядів, зверху над кожною тетрадою записано порядковий номер тетради, також для зручності візуального сприйняття між окремими тетрадами зроблено відступи:



В тетраді № 5 відсутні два старші розряди. В разі необхідності — на позицію пустих старших розрядів, можна записати незначущі нулі, які не вплинути на величину числового значення:



Тепер кожній тетраді ставиться у відповідність шістнадцятковий символ (четирьохроздядні двійкові коди для відповідних шістнадцяткових символів показано в табл. 2.9). Запишемо наступний крок перетворення так:



Отже, запишемо остаточний результат таким чином:

$$111010011011110001_2 = 3A6F1_{16}$$

При перетворенні значення із шістнадцяткової системи числення в двійкову систему числення, необхідно виконати зворотні дії — кожному шістнадцятковому символу поставити у відповідність свій четирьохроздядний двійковий код із табл. рис. 2.9, записати отриману двійкову кодову послідовність, що і буде остаточним результатом. Покажемо це на прикладі. Нехай потрібно перетворити в двійкову систему числення такий шістнадцятковий код:

$$5C0BE80_{16}$$

Для кожного шістнадцяткового символу запишемо його відповідний чотирьохроздний двійковий код. Для зручності — між окремими шістнадцятковими розрядами зроблено відступи:

5_{16}	C_{16}	0_{16}	B_{16}	E_{16}	8_{16}	0_{16}
$\overbrace{0101}$	$\overbrace{1100}$	$\overbrace{0000}$	$\overbrace{1011}$	$\overbrace{1110}$	$\overbrace{1000}$	$\overbrace{0000}$

Двійковий код буде таким (для зручності візуального сприйняття між окремими тетрадами поставлено пробіл):

$$0101\ 1100\ 0000\ 1011\ 1110\ 1000\ 0000_2$$

В разі необхідності можна старші незначущі нулі не записувати (в даному випадку значення 0 в найстаршому розряді не впливає на величину числового значення, і якщо не вказано ніяких обмежень щодо вимог запису результату, цей незначущий нуль можна опустити). В даному випадку при записі остаточного результату незначущий нуль в найстаршому розряді в двійковому коді залишений, тому запис буде такий:

$$5C0BE80_{16} = 0101\ 1100\ 0000\ 1011\ 1110\ 1000\ 0000_2$$

Описана методика швидкого перетворення дозволяє без будь-яких зусиль виконати перетворення значення, наприклад, із двійкової системи числення у шістнадцяткову систему числення або із вісімкової системи числення у двійкову систему.

У випадку якщо потрібно перетворити значення із шістнадцяткової системи числення у вісімкову систему числення (або навпаки), то необхідно лише, застосувавши описану методику, спочатку отримати значення в двійковій системі числення, а потім виконати перетворення до потрібної системи числення.

2.5 Прямий код. Представлення беззнакових та додатних цілих чисел в пам'яті комп'ютера

При написанні програм для персональних комп'ютерів або для вбудованих систем на мові програмування С, виникає необхідність оперувати різними типами даних. Важливо розуміти яким чином значення відповідного типу даних зберігається в пам'яті комп'ютера. В комп'ютерній техніці та в цифровій техніці загалом, дані в пам'яті електронно-обчислювальної машини зберігаються у вигляді двійкових кодів. Однак різні типи даних зберігаються по-різному, тобто для різних типів даних може бути застосована своя відповідна процедура перетворення певних даних, наприклад, десяткових чисел в двійковий код [1, 2, 4].

В мові програмування С існують різні типи даних, які призначенні для зберігання різного роду інформації. Загалом, типи даних, які призначенні для зберігання числової інформації, можна розділити на відповідні категорії. Спрощена класифікація показана на рис. 2.5.

Виходячи із зазначененої схеми, видно, що дійсні числа (числа, які характеризуються наявністю цілої та дробової частини, наприклад, +3.14, -5.125, +2.0), зберігаються у відповідному форматі, який називається формат з плаваючою комою (або плаваючою точкою) або ще по-іншому називають формат з рухомою комою (або рухомою точкою), — в залежності від того який символ використовується для розділення цілої та дробової частини — кома чи точка.

Для зберігання цілих чисел використовується формат з *фіксованою* комою (точкою). Цілі числа можна розділити на знакові числа (тобто такі, що крім величини ще зберігають знак числа — «+» або «-») та беззнакові числа (цілі числа, які не мають знаку, тобто знак числа відсутній, зберігається тільки величина. Це може бути сприйнято як певний аналог модуля числа). Треба пам'ятати, що *додатні цілі* числа та *беззнакові цілі* числа зберігаються в *прямому* коді, а *від'ємні цілі* числа — зберігаються в *доповнальному* коді [2, 4].



Рис. 2.5 — Спрощена класифікація типів даних із зазначенням виду двійкового коду (прямий код чи доповняльний код), який застосовується для зберігання числових даних в пам'яті комп'ютера

Розглянемо збереження додатних цілих чисел та беззнакових цілих чисел в прямому коді.

Пряний код отримується шляхом безпосереднього перетворення числа в двійковий код відповідно до методик, які були розглянуті вище.

Беззнакові типи даних, які призначені для зберігання беззнакових цілих чисел, не містять інформації про знак тому що знак відсутній. Для зберігання двійкового коду певного числа в пам'яті комп'ютера виділяється певна кількість двійкових розрядів (певна кількість бітів). Наприклад, в мові програмування С для типу даних **unsigned char**, який призначений для зберігання беззнакових цілих чисел, виділено 8 розрядів (8 бітів). 8 бітів складають 1 байт. Таким чином, числове значення, для зберігання якого обрано тип даних **unsigned char**, в пам'яті комп'ютера буде займати 1 байт. Схематично комірку пам'яті, яка має розмір 1 байт і призначена для зберігання значень типу **unsigned char** можна представити наступним чином (зверху над відповідним розрядом вказується номер двійкового розряду):

7	6	5	4	3	2	1	0

Для беззнакових типів даних (в даному випадку для беззнакового типу даних **unsigned char**) всі розряди розрядної сітки призначені для зберігання величини числа. Виходячи із цього, мінімальне значення, яке може бути збережене в наведеній розрядній сітці розміром 1 байт, — це значення 0 (нуль). Число 0 (нуль) може бути збережено у вказаній розрядній сітці таким чином:

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0

Найбільше значення типу даних **unsigned char**, яке може бути збережене в комірці пам'яті розміром 1 байт, становить 255_{10} . Числу 255_{10} відповідає **пряний код**, що представляє собою послідовність із восьми двійкових одиниць: $255_{10} = 11111111_2$:

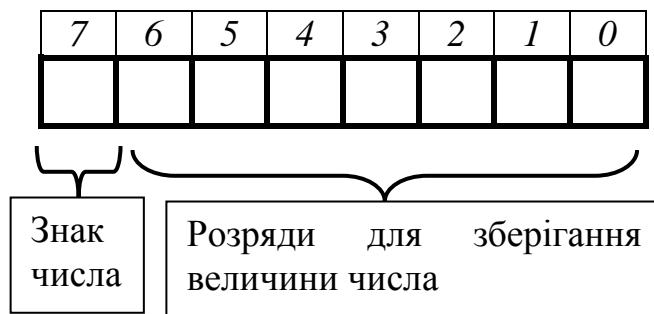
7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1

Наприклад, значення 12_{10} типу **unsigned char** буде збережено в комірку пам'яті розміром 1 байт таким чином (враховуючи, що $12_{10} = 1100_2$):

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0

При записі двійкового коду в задану комірку пам'яті (в задану розрядну сітку) важливо звертати увагу на правильне розміщення відповідних розрядів (наймолодший розряд — має номер 0, найстарший розряд — має номер 7).

Знаковий цілий тип даних передбачає крім величини числа також зберігати інформацію про знак. В мові програмування С знаковий цілий тип даних для зберігання значень якого виділяється 1 байт пам'яті називається **char**. В цьому випадку комірка пам'яті розміром 1 байт буде використовувати розряди з номерами 0...6 для зберігання величини числа, а розряд з номером 7 буде призначений для зберігання знаку числа. Схематично, таку комірку пам'яті можна представити наступним чином:



Знаку «+» відповідає значення 0_2 , що записується в найстаршому розряді. Додатні числа зберігаються в **прямому коді**.

Знаку «-» відповідає значення 1_2 , що записується в найстаршому розряді. Від'ємні числа зберігаються в **доповняльному коді** [1, 2, 4].

Наприклад, значення $+25_{10}$ — буде зберігатися в пам'яті комп'ютера в *прямому* коді. Оскільки число додатне, то в найстарший розряд (розряд із номером 7) буде записано значення 0₂. А розряди з номерами від 0 по 6 — будуть призначені для зберігання двійкового коду, який отримується шляхом безпосереднього перетворення числа 25_{10} в двійкову систему числення. Значення $25_{10} = 11001_2$, тому комірка пам'яті розміром 1 байт, яка була виділена для зберігання значення типу **char**, буде зберігати таку двійкову кодову послідовність:



Таким чином, можна зробити висновок, що найбільше додатне значення, яке може бути збережено в типі даних **char** буде визначатися такою двійковою послідовністю (записаною в прямому коді):



Представленому в розрядній сітці двійковому коду буде відповідати значення $+127_{10}$.

Для перетворення двійкової кодової послідовності, яка записана в прямому коді, в десяткову систему числення необхідно застосувати відповідну методику, яка була описана вище — необхідно кожен біт помножити на свій ваговий коефіцієнт (двійка у відповідній степені, де показник степені визначається номером відповідного розряду), і знайти суму отриманих добутків.

Від'ємні числа в *прямому* коді в пам'яті комп'ютера не зберігаються !!!

2.6 Доповняльний код. Представлення від'ємних цілих чисел в пам'яті комп'ютера

Від'ємні цілі числа в пам'яті комп'ютера зберігаються в *доповняльному коді*. Для того щоб виконати перетворення від'ємного цілого числа із десяткової системи числення в доповняльний код, необхідно виконати наступні кроки [2, 4]:

1. Отримати *прямий код модуля* від'ємного числа (для цього необхідно виконати перетворення модуля від'ємного числа із десяткової системи числення в двійкову систему числення згідно із методикою, яка була описана в розділі 2.1). Якщо передбачається записувати результат в розрядну сітку вибраного розміру, то необхідно в такому разі заповнювати всі розряди, наприклад, заповнювати незначущими нулями старші розряди.
2. Виконати інверсію розрядів в двійковому коді, який отримано в пункті 1.
3. До двійкового коду, який отримано в пункті 2, додати 1. Виконавши додавання, отримати **доповняльний код**.

Розглянемо на прикладі описаний алгоритм. Нехай задано число -17_{10} . Для збереження даного значення обрано тип даних **char** розміром 1 байт. Необхідно знайти доповняльний код для числа -17_{10} , і результат записати в комірку пам'яті. Для цього необхідно:

1. Знайти прямий код для модуля від'ємного числа і записати результат в восьмирозрядну сітку: $|-17_{10}| = 17_{10} = 00010001_2$:

7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1

2. Виконати інверсію розрядів — одиниці замінити на нулі, а нулі замінити на одиниці:

7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0

3. Отримати доповняльний код, додавши одиницю до молодшого розряду (додавання виконується аналогічним чином як і при додаванні в стовпчик при операції значеннями в десятковій системі числення:

7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0
+							
							1
1	1	1	0	1	1	1	1

Таким чином, **доповняльний код** для значення -17_{10} буде таким:

7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	1
Знак числа							

Однація в найстаршому розряді (в знаковому розряді) вказує на те, що число, яке зберігається в комірці пам'яті — від'ємне.

Для того, щоб виконати перетворення значення із доповняльного коду в десяткову систему числення, необхідно виконати наступні кроки:

1. Виконати інверсію розрядів.
 2. Додати одиницю до двійкового коду, що отриманий на попередньому кроці.
 3. Двійковий код, що отримано на попередньому кроці, перетворити в десяткове число. Отримане значення буде *модулем* від'ємного числа.
- При записі остаточного результату необхідно поставити знак «мінус».

Розглянемо на прикладі алгоритм перетворення із доповняльного коду в десяткову систему числення. Нехай задано наступний доповняльний код:

7	6	5	4	3	2	1	0
1	1	1	1	0	0	0	0

Виконуємо інверсію розрядів, отримаємо:

7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1

Додаємо одиницю до молодшого розряду і виконуємо додавання аналогічним чином як це відбувалося при додаванні в стовпчик десяткових чисел:

+	$\begin{array}{ccccccccc} & 1 & 1 & 1 & 1 \\ & \swarrow & \searrow & \swarrow & \searrow \\ \hline \end{array}$	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	1	1
0	0	0	1	0	0	0	0	0	0

До молодшого розряду (який має номер 0) додаємо одиницю:

$$1_2 + 1_2 = 2_{10} = 10_2$$

Таким чином, виконавши додавання отримали двійковий код 10_2 . Молодший розряд — 0_2 записується в комірку для результату (на позицію із номером 0), а старший розряд — 1_2 переноситься по розрядній сітці і додається до розряду із номером 1, отже маємо:

$$1_2 + 1_2 = 2_{10} = 10_2$$

Знову отримано двійковий код 10_2 . Молодший розряд — 0_2 записується в комірку для результату (на позицію із номером 1), а старший розряд — 1_2 переноситься по розрядній сітці і додається до розряду із номером 2, отримаємо:

$$1_2 + 1_2 = 2_{10} = 10_2$$

Отримано двійковий код 10_2 . Молодший розряд — 0_2 записується в комірку для результату (на позицію із номером 2), а старший розряд — 1_2 переноситься по розрядній сітці і додається до розряду із номером 3. Будемо мати:

$$1_2 + 1_2 = 2_{10} = 10_2$$

В результаті знову отримали двійковий код 10_2 . Молодший розряд — 0_2 записується в комірку для результату (на позицію із номером 3), а старший розряд — 1_2 переноситься по розрядній сітці і додається до розряду із номером 4. Маємо:

$$1_2 + 0_2 = 1_2$$

Отриманий результат записується в комірку для зберігання результату на позицію із номером 4.

Більше переходу одиниці по розрядній сітці не відбувається, тому в старші розряди переписуються значення розрядів, що записані на відповідних позиціях в першому доданку.

Остаточний двійковий код такий:

7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0

Виконавши перетворення отриманого двійкового коду в десяткову систему числення отримаємо:

$$1 \times 2^4 = 16$$

Пам'ятаючи, що отримане значення — це модуль від'ємного числа, тому при записі остаточної відповіді обов'язкового необхідно записати знак «мінус» перед десятковим числом:

$$1111000_2 = -16_{10}$$

2.7 Формат з фіксованою комою

Формат з фіксованою комою (або, по-іншому, формат з фіксованою точкою) передбачає встановлення десяткової коми (десяткової точки), яка розділяє цілу та дробову частину числа, в певній позиції в розрядній сітці [2, 4]. Наприклад, в загальному варіанті розрядна сітка, яка призначена для зберігання значення в форматі з фіксованою комою, в залежності від розташування цієї десяткової коми (десяткової точки), може мати такий вигляд: десяткова кома (десятикова точка) встановлюється в середині розрядної сітки. Таким чином, в загальному випадку, в розрядній сітці окремий двійковий розряд повинен бути призначений для зберігання знаку числа, частина двійкових розрядів призначена для зберігання цілої частини числа, а остання частина двійкових розрядів використовується зберігання дробової частини. Схематично, таку розрядну сітку показано на рис. 2.6.

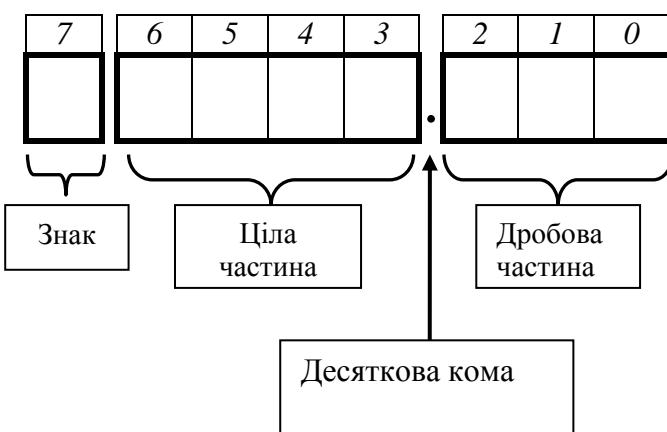


Рис. 2.6 — Розрядна сітка з виділеними розрядами для зберігання цілої та дробової частини числа в двійковій системі числення

Використаємо представлену на рис. 2.6 розрядну сітку для ілюстрації переведення чисел, що містять цілу та дробову частину із десяткової системи числення в двійкову систему числення. Для зручності виконання зворотного перетворення, змінимо номери розрядів, які записані над комірками, що призначенні для зберігання відповідних бітів. Розряди цілої частини будуть нумеруватися починаючи з 0, де 0 — це номер наймолодшого розряду цілої

частини, який стоїть зліва від десяткової коми (десяткової точки). Розряди дробової частини будуть нумеруватися починаючи з -1 , де номер -1 буде мати старший розряд дробової частини, який стоїть справа від десяткової коми (десяткової точки). Виходячи із цього, розрядна сітка із зміненою нумерацією розрядів буде такою:

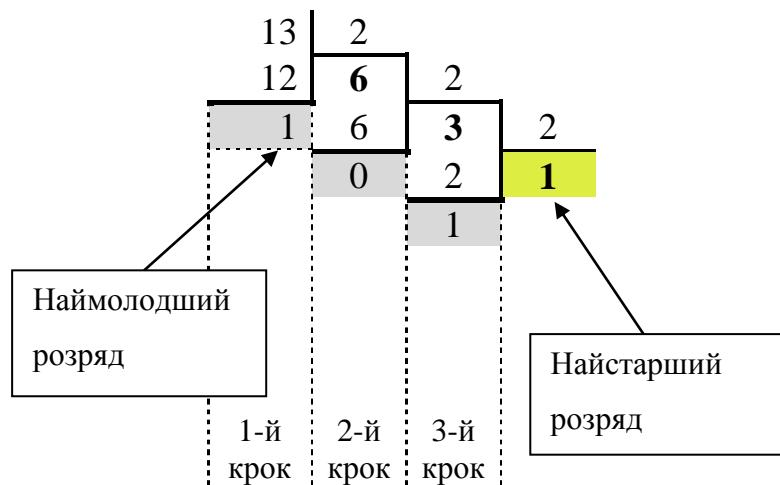
4	3	2	1	0	-1	-2	-3

В якості прикладу виконаємо перетворення десяткового числа $+13.875_{10}$.

Спочатку треба виконати початковий аналіз вхідного значення, яке необхідно перетворити в двійкову систему числення:

1) число $+13.875_{10}$ додатне, отже в знаковий розряд (розряд із номером 4 відповідно до схематичного представлення розрядної сітки показаної вище) буде записано 0_2 .

2) ціла частина числа дорівнює 13_{10} . Враховуючи, що число додатне, то в двійковій системі числення воно буде зберігатися в прямому коді. Щоб отримати прямий код для цілої частини десяткового числа, потрібно, виконати послідовно операцію ділення на 2 та отримати двійковий код:



Отже, прямий код для числа 13_{10} буде таким:

$$13_{10} = 1101_2$$

3) Необхідно виконати перетворення до двійкової системи числення значення дробової частини початкового десяткового числа. Отже, необхідно отримати двійковий код для значення 0.875_{10} . Для цього потрібно послідовно виконувати операцію множення на 2.

На першому кроці після знаходження добутку — значення цілої частини отриманого результату буде виступати в якості найстаршого двійкового розряду дробової частини (розряд номер -1 в розрядній сітці). На другому кроці — береться дробова частина результату із попереднього кроку і множиться на 2. Ціла частина результату — це другий двійковий розряд дробової частини (розряд номер -2 в розрядній сітці). На третьому кроці — береться результат дробової частини попереднього кроку і множиться на 2. Ціла частина результату — це третій двійковий розряд дробової частини (розряд номер -3 в розрядній сітці). Якщо розрядна сітка, що призначена для зберігання дробової частини має більшу кількість розрядів, ніж в представлений на рис. 2.6 схемі, то описаний процес продовжується поки всі розряди не будуть заповнені. Якщо на певному етапі обрахунків дробова частина результату після чергового множення на 2 буде нульовою — на цьому цей процес припиняється. Якщо при цьому залишаються незаповненими частина розрядів дробової частини — на відповідні позиції записують нулі.

В наведеному прикладі, необхідно перетворити значення 0.875_{10} в двійковий код:

- 1) $0.875 \times 2 = 1.75$ (в розряд з номером -1 буде записано 1_2)
- 2) $0.75 \times 2 = 1.5$ (в розряд з номером -2 буде записано 1_2)
- 3) $0.5 \times 2 = 1.0$ (в розряд з номером -3 буде записано 1_2)

Таким чином, маємо:

$$0.875_{10} = 0.111_2$$

Загалом, отримаємо такий результат:

$$+13.875_{10} = +1101.111_2$$

І в розрядній сітці даний результат буде збережений в такому вигляді:

4	3	2	1	0	-1	-2	-3
0	1	1	0	1	1	1	1

Для виконання зворотного перетворення (враховуючи, що в даному прикладі використовується прямий код, оскільки вхідне десяткове число — додатне), необхідно двійкові розрядки цілої частини помножити на 2 у відповідній степені, знайти суму добутків, і отримати значення цілої частини числа в десятковій системі числення. Теж саме потрібно зробити із двійковими розрядами дробової частини числа, і тим самим отримавши значення дробової частини в десятковій системі числення. Виконавши ці дії, можна отримати значення числа в десятковій системі числення:

$$\text{ЦілаЧастина}_{10} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 0 + 1 = 13_{10}$$

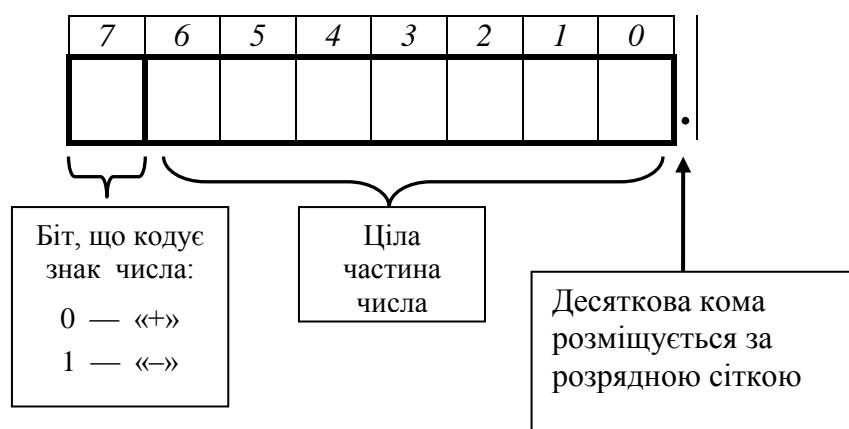
$$\text{ДробоваЧастина}_{10} = 1 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 0.5 + 0.25 + 0.125 = 0.875_{10}$$

Таким чином, остаточний результат, отриманий шляхом перетворення із двійкової системи числення в десяткову систему числення:

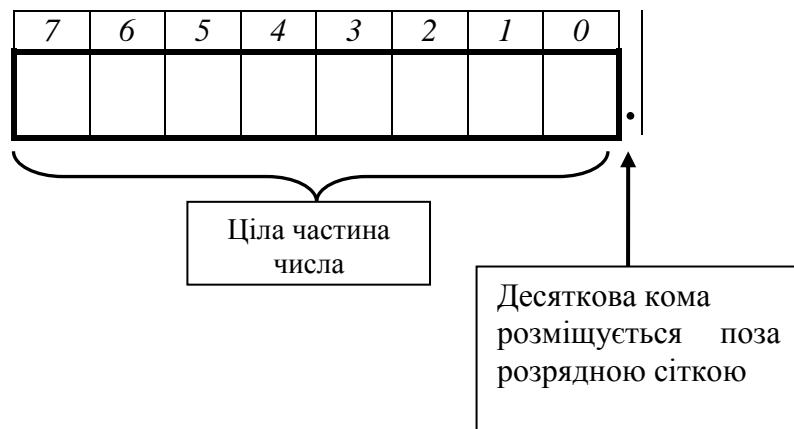
$$X_{10} = +13.875_{10}$$

Представлений вище варіант формату з фіксованою комою, тобто коли одні розряди виділяються для зберігання цілої частини числа, а інша група розрядів використовується для зберігання дробової частини числа, — часто не використовується на практиці. Замість цього використовується варіант формату з фіксованою комою (фіксованою точкою), коли десяткова кома (десяткова точка), яка відділяє розряди цілої та дробової частини, розміщується поза розрядною сіткою таким чином, що розрядів для зберігання дробової частини немає. Схематично це показано на рис. 2.7 для випадку знакового типу даних (коли окремий розряд призначений для зберігання знаку числа), і для випадку беззнакового типу даних (коли всі розряди призначенні для зберігання величини числа). В такому разі десяткова кома уявно знаходитьсь за розрядною сіткою біля розряду із номером 0, тому місця для зберігання дробової частини числа не передбачено. Таким чином, формат з фіксованою комою

використовується для зберігання лише цілих чисел — знакових або беззнакових. При намаганні зберегти дійсне число, що містить цілу та дробову частину — дробова частина відсікається, і втрачається, а зберігається лише ціла частина. Тобто не відбувається ніякого округлення дійсного числа до найближчого цілого, а виконується відкидання дробової частини. Про це важливо пам'ятати на етапі обрання відповідних типів даних при написанні програм на мові програмування. С.



a)



б)

Рис. 2.7 — Формат з фіксованою комою для знакового цілого типу даних *char* (а) та беззнакового цілого типу даних *unsigned char* (б)

2.8 Формат з плаваючою комою для зберігання дійсних чисел

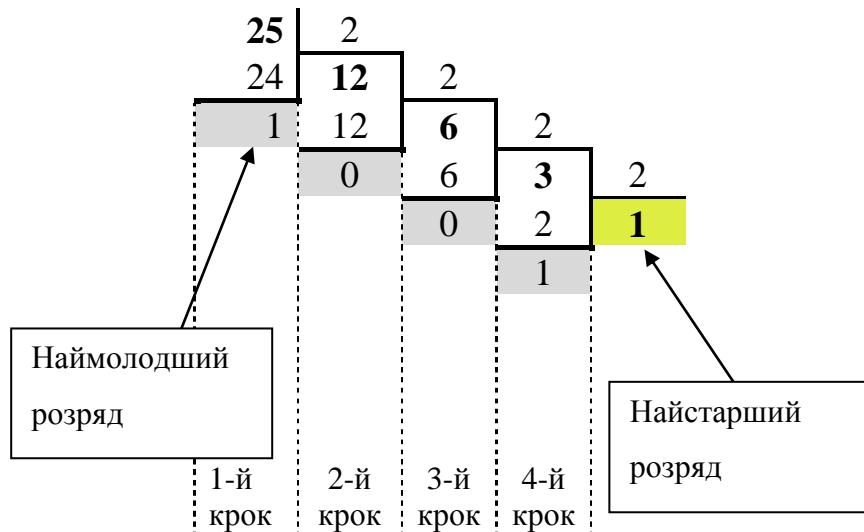
Для зберігання в пам'яті комп'ютера дійсних чисел, тобто таких, що містять цілу та дробову частину, використовують формат з плаваючою комою (плаваючою точкою) або ще можна зустріти в електронних джерелах в Інтернеті іншу назву — формат з рухомою комою (рухомою точкою). В мові програмування С використовуються різні типи даних, які призначені для зберігання дійсних чисел в форматі з плаваючою комою, яких об'єднує одинаковий принцип перетворення дійсних чисел в двійкову систему числення, а відрізняються різні типи даних — кількістю розрядів, яіа виділяються для зберігання відповідних складових дійсного числа в двійковому коді, що в свою чергу впливає на точність збереження чисел, а також на діапазон можливих значень, які можуть зберігатися у відповідному типі даних. Розглянемо принцип перетворення дійсного десяткового числа, що містить цілу та дробову частини із десяткової системи числення у двійковий код, який буде відповідати формату з плаваючою комою одинарної точності, що відповідає в мові програмування С типу даних *float* [1, 2, 4].

В якості прикладу, на основі якого буде продемонстровано технологію перетворення значення із десяткової системи числення в двійкову систему числення формату з плаваючою комою одинарної точності, в якому в пам'яті комп'ютера зберігаються значення типу даних *float*, візьмемо таке десяткове число:

$$-25.6875_{10}$$

Необхідно виконати ряд кроків:

1. Перетворити цілу частину числа із десяткової системи числення в двійкову систему числення, при цьому на етапі перетворення знак числа до уваги не брати:



Таким чином, маємо наступне:

$$25_{10} = 11001_2$$

2. Перетворити дробову частину числа із десяткової системи числення в двійкову систему числення. Для заданого прикладу, необхідно перетворити значення 0.6875_{10} в двійковий код:

- 1) $0.6875 \times 2 = 1.375$
- 2) $0.375 \times 2 = 0.75$
- 3) $0.75 \times 2 = 1.5$
- 4) $0.5 \times 2 = 1.0$

Отже, маємо:

$$0.6875_{10} = 0.1011_2$$

3. Перетворення числа в нормалізовану форму.

Після виконання перших двох кроків отримано двійковий код для вхідного десяткового числа:

$$-25.6875_{10} = -11001.1011_2$$

Після цього отриману двійкову кодову послідовність необхідно перетворити в нормалізовану форму.

Нормалізована форма передбачає запис двійкового числа в такому форматі [2]:

$$\pm 1.M \times q^{\pm E}$$

де « \pm » — позначає знак числа («+» для додатного числа або «-» для від'ємного); $1.M$ — мантиса числа; M — хвіст мантиси; q — основа системи числення (для двійкової системи числення $q=2$); $\pm E$ — показник степеня (для зручності записаний в десятковій системі числення), де символом « \pm » позначається значення показника степеня.

Для отриманого значення -11011.1011_2 нормалізована форма числа буде мати такий вигляд:

$$-11001.1011_2 = -1.10011011 \times 2^{+4}$$

Таким чином, маємо:

$$M = 10011011$$

$$E_{10} = +4_{10}$$

4. Обрахунок зміщеного порядку.

Зміщений порядок (позначимо цю величину символом Z) визначається шляхом додавання константи 127_{10} до значення степеня E (яке було визначено на попередньому кроці). (Константа 127_{10} використовується при перетворенні в двійковий код значення типу **float !!!**)

$$Z_{10} = E_{10} + 127_{10}$$

Для розглянутого прикладу:

$$Z_{10} = 4_{10} + 127_{10} = 131_{10}$$

Необхідно виконати перетворення значення зміщеного порядку в двійкову систему числення:

$$131_{10} = 10000011_2$$

Отже, значення зміщеного порядку в двійковій системі числення буде таким:

$$Z_2 = 10000011_2$$

5. Записати код знаку числа, а також хвіст мантиси (M) та значення зміщеного порядку (Z_2) в розрядну сітку, яка складається із 32 розрядів. Схематично така розрядна сітка показана на рис. 2.8 [1, 2].

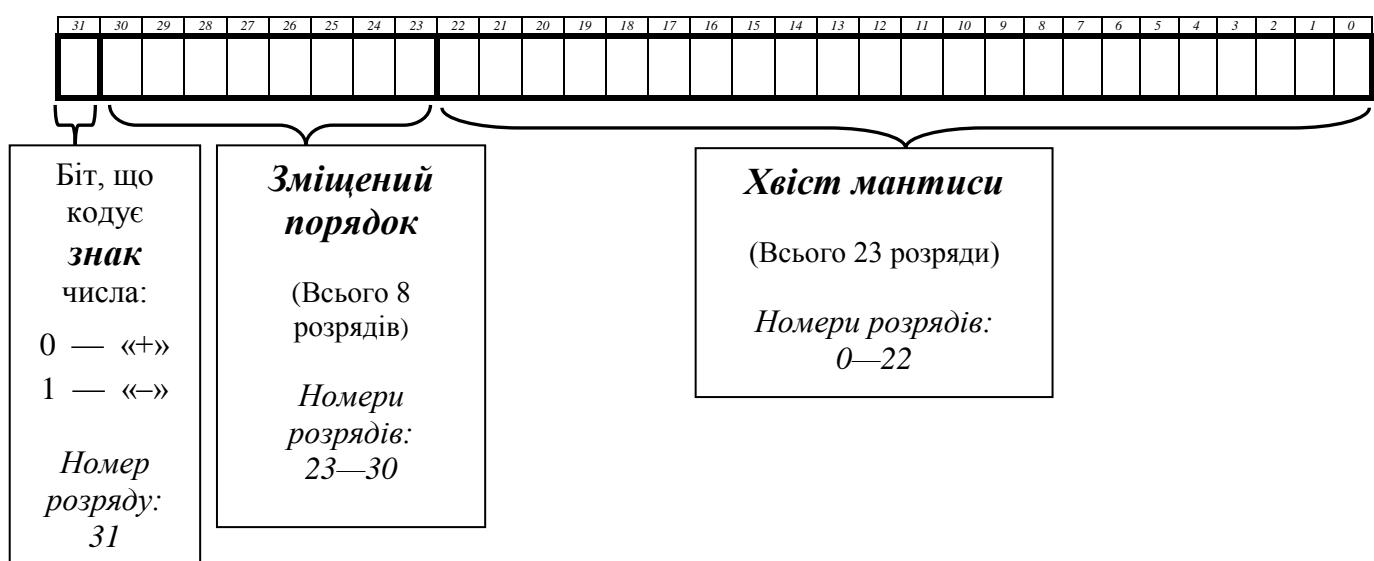


Рис. 2.8 — Розрядна сітка для зберігання значення типу даних float (формат з плаваючою комою одинарної точності)

Таким чином, виконавши описані перетворення і отримавши відповідні двійкові кодові послідовності, які будуть записуватися у відповідні поля в розрядній сітці, остаточний результат для значення -25.6875_{10} буде представлений наступним чином (пусті розряди, які призначені для зберігання хвоста мантиси будуть заповнюватися нулями):

Для компактності запису, можна використати шістнадцяткову систему числення. Для цього необхідно виділити тетради (групи по чотири розряди), і

поставивши у відповідність кожній тетраді значення шістнадцяткового символу, отримати шістнадцятковий код:

3 1	3 0	2 9	2 8	2 7	2 6	2 5	2 4	2 3	2 2	2 1	2 0	1 9	1 8	1 7	1 6	1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	1	1	1	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{16}				1_{16}				C_{16}				D_{16}				8_{16}				0_{16}				0_{16}							

Таким чином, можемо записати остаточний результат перетворення (при записі двійкового коду для зручності сприйняття між окремими тетрадами поставлено пробільний символ):

$$-25.6875_{10} = 1100\ 0001\ 1100\ 1101\ 1000\ 0000\ 0000\ 0000_2 = C1CD8000_{16}$$

Для того щоб виконати перетворення двійкового коду, який записаний в 32-х розрядній сітці, необхідно виконати перетворення в зворотному порядку [2].

Або можна скористатися наступною розрахунковою формулою:

$$X_{10} = (-1)^S \times 2^{Z_{10} - 127_{10}} \times \left(1 + \frac{m_{10}}{2^{23}}\right),$$

де $S=1$, якщо знаковий біт дорівнює 1; $S=0$, якщо знаковий біт дорівнює 0;
 Z_{10} — зміщений порядок, записаний в десятковій системі числення (необхідно взяти двійкові розряди з 23 по 30, і перетворити утворений ними двійковий код в десяткову систему числення);

m_{10} — число, що отримане шляхом перетворення двійкового коду, записаного в розрядах з 0 по 22, в десяткову систему числення.

Для заданого прикладу, маємо:

$S = 1$ (оскільки в знаковому розряді, що має номер 31, стоїть 1₂);

$$Z_{10} = 1 \times 2^7 + 1 \times 2^1 + 1 \times 2^0 = 128_{10} + 2_{10} + 1_{10} = 131_{10}$$

$$m_{10} = 1 \times 2^{22} + 1 \times 2^{19} + 1 \times 2^{18} + 1 \times 2^{16} + 1 \times 2^{15} = 4194304 + 524288 + 262144 + 65536 + 32768 = 5079040$$

$$\begin{aligned}
 X_{10} &= (-1)^1 \times 2^{131_{10}-127_{10}} \times \left(1 + \frac{5079040}{2^{23}}\right) = \\
 &= -1 \times 2^4 \times \left(1 + \frac{5079040}{8388608}\right) = -16 \times (1 + 0.60546875) = \\
 &= -16 \times 1.60546875 = -25.6875_{10}
 \end{aligned}$$

Отже, виконавши зворотне перетворення, скориставшись для цього розрахунковою формулою, отримано початкове значення числа в десятковій системі числення, що дозволяє стверджувати про правильність виконання відповідних операцій по перетворенню значення із двійкового коду, записаного у форматі з плаваючою комою одинарної точності, в десяткову систему числення.

2.9 Питання для самоконтролю

1. Виконайте перетворення числа -14_{10} в двійкову систему числення.
2. Яким чином можна виконати перетворення значення із двійкової системи числення у вісімкову систему числення?
3. Виконайте перетворення значення $A0FE8100_{16}$, що зберігається у форматі з плаваючою комою одинарної точності, в десяткову систему числення.
4. Якщо для зберігання знакових цілих чисел обрано тип даних, що складається із 4 двійкових розрядів. Визначте діапазон числових значень, що представлені в десятковій системі числення, які можуть бути коректно збережені в пам'яті комп'ютера.
5. Обрано тип даних `char`, який має розмір 1 байт, і призначений для зберігання знакових цілих чисел. Що відбудеться при намаганні зберегти десяткове число 267_{10} ? Яке значення буде збережене в пам'яті комп'ютера?
6. Виконайте перетворення шістнадцяткової кодової послідовності $2FE10_{16}$ в десяткову систему числення.
7. Послідовність 10011000_2 записана в доповняльному коді. Виконайте перетворення її в десяткову систему числення.

3. СТРУКТУРА ПРОГРАМИ НА МОВІ С. ТИПИ ДАНИХ. ОПЕРАЦІЙ

3.1 Структура програми на мові програмування С

Програма, що написана мовою програмування С, представляє собою текст, що містить в собі набір препроцесорних директив, які, наприклад, дозволяють включати до програми вміст заголовочних файлів стандартної бібліотеки, крім того програма може містити: константи; запис прототипів функцій, виклики функцій; описи функцій, які в програмі створює розробник програми та ін. Текст програми на мові С може бути записаний в одному файлі, але також можуть бути створені проекти, в яких текст програми зберігається в багатьох файлах. Невеликі за обсягом проекти можуть бути реалізовані шляхом написання тексту програми в одному файлі. Спрощена структура такої програми схематично представлена на рис. 3.1 [3]:

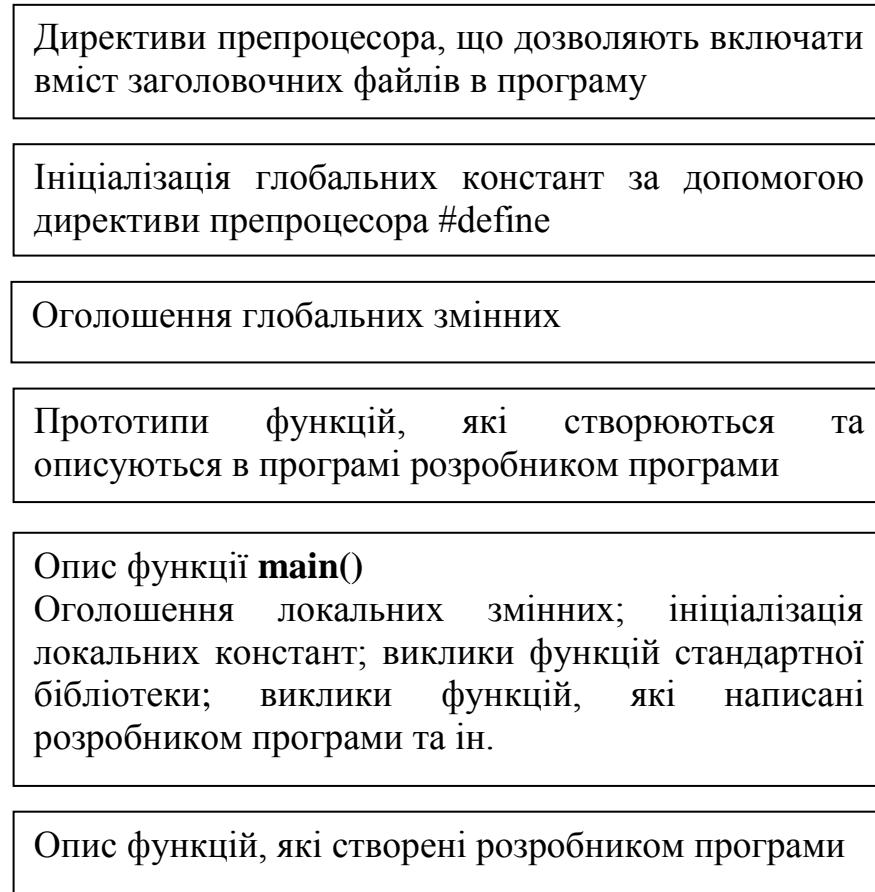


Рис. 3.1 — Схематичне представлення простої програми на мові С

Препроцесор — це програма, яка виконує обробку тексту програмного коду, написаного розробником на мові програмування, перед тим як виконувати компіляцію тексту програми відповідним компілятором.

Компілятор — це програма, яка виконує перетворення програмного коду, який написаний розробником на мові програмування, в набір команд, що виконуються електронно-обчислювальною машиною.

Директива препроцесора — це спеціальна команда, яка починається із символу «решітка» — #. Директива препроцесора використовується для попередньої обробки коду програми перед тим, як код програми буде компілюватися.

Наприклад, директива препроцесора `#include` дозволяє помістити вміст заголовочного файлу в текст програми, яка пишеться розробником.

Заголовочний файл — може містити прототипи функцій, значення констант, макроси тощо.

Прототип функції — це оголошення (декларація) функції, в якому вказується назва функції; кількість та типи даних аргументів, які функція отримує при виклику її в програмі; тип даних результату, що функція повертає. При цьому, в записі прототипу не приводиться опис функції, тобто не записується програмний код, який повинен виконуватися при виклику функції. Прототип функції також іноді називають інтерфейсом функції. Тобто прототип функції не розкриває внутрішню реалізацію функції, а лише надає інформацію про те як функція називається (при виклику функцій в програмі необхідно вказувати її ім'я), а також прототип надає інформацію про те які параметри функція може отримувати, і який тип даних результату, що функція повертає (функція може повертати певний результат, а може його не повертати).

Мова програмування С передбачає наявність стандартної бібліотеки функцій, які охоплюють різні сфери обробки даних. Це можуть бути математичні функції, наприклад, які дозволяють обраховувати логарифм, модуль, степінь та ін.; це можуть бути функції, які дозволяють виводити інформацію на екран монітора в консольне вікно, або у файл на жорсткому

диску; це можуть бути функції, які дозволяють зчитувати інформацію з клавіатури або з файлу та ін. Прототипи відповідних функцій записані у відповідних заголовочних файлах. Таким чином, якщо виникає необхідність, наприклад, в програмі використовувати математичні функції стандартної бібліотеки мови С, тоді потрібно в програму включити відповідний заголовочний файл. Якщо передбачається введення даних з клавіатури і виведення результатів на екран в консольне вікно, тоді потрібно включити в програму відповідний заголовочний файл, що містить прототипи функцій, які потрібні для зчитування даних з клавіатури та виведення результату на екран монітору. Таким чином, підключення заголовочного файла дозволяє надати компілятору всю необхідну інформацію про функції, які можуть використовуватися в програмі при її написанні.

Заголовочні файли стандартної бібліотеки підключаються в програму за допомогою директиви препроцесора `#include`, яка записується на початку програми, і має такий формат запису [3, 4]:

```
#include <им'я заголовочного файла.разширення файла>
```

Наприклад, для того щоб використовувати в програмі математичні функції стандартної бібліотеки, необхідно підключити заголовочний файл `math.h`. Це можна зробити за допомогою директиви препроцесора `#include` наступним чином:

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

Якщо передбачається, що в програмі будуть використовуватися функції по зчитуванню значень з клавіатури та виведення даних на екран монітору в консольне вікно — для цього можна скористатися відповідними функціями, прототипи яких зберігаються в заголовочному файлі `stdio.h`, тоді на початку програми на окремому рядку необхідно додати запис директиви препроцесора:

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

При реалізації в програмі відповідних алгоритмів, часто виникає необхідність оперувати константами. Константи задаються в програмі на етапі написання коду програми. Константи не можуть змінювати своє значення на етапі виконання програми. Константи можуть бути іменовані (тобто такі, які мають ім'я) та неіменовані (наприклад, якесь число, що використовується при обрахунку деякого запрограмованого в програмі математичного виразу). Константи можуть бути глобальними та локальними. Глобальні константи задаються на початку програми і можуть використовуватися будь-де нижче в тексті програми (для випадку, коли текст програми записується в одному файлі). Один із способів задати в програмі глобальну константу — це скористатися директивою препроцесора `#define`. Формат такого запису наступний [3—8]:

#define [пробіл] **Ідентифікатор** [пробіл] **Текст**

В наведеному прикладі директиви препроцесора `#define` — це обов'язковий елемент, який записується на окрему рядку, після слова `#define` ставиться символ «пробіл» шляхом натискання відповідної клавіші на клавіатурі, потім записується ідентифікатор (ідентифікатор — це по суті ім'я константи, яке може обиратися довільно, але при цьому необхідно додержуватися встановлених правил, які накладають певні обмеження на формування назви ідентифікаторів). Після ідентифікатора ставиться символ «пробіл», а вже після цього записується **Текст**, який по-суті виступає в якості значення константи. Наприклад, для того, щоб створити константу, для якої обрано ім'я `N`, і для якої встановлюється значення рівне `15`, необхідно записати так:

`#define N 15`

Перед виконанням компіляції коду програми, препроцесор виконує зміни в тексті програми — в програму додається вміст тих заголовочних файлів, які підключенні в програму за допомогою директиви `#include`, а також відбувається заміщення в тексті програми тих ідентифікаторів, які записані вище в директиві

`#define`, на відповідні їм значення, які записані на місці Текст в наведеному вище шаблоні запису. Наприклад, при записі `#define N 15` всюди (окрім коментарів) де буде використовуватися ідентифікатор `N` в текст програми буде вставлено значення `15`. Така заміна є справедливою по відношенню до окремих лексем, ім'я яких співпадають із ідентифікатором, який записано в рядку директиви препроцесора після слова `#define`.

Лексема — це певна самостійна одиниця тексту програми, яка не може бути розділена на дрібніші елементи. Наприклад, до лексем відносяться ідентифікатори, службові зарезервовані слова мови програмування С, числові константи тощо.

Глобальні змінні на відмінну від глобальних констант можуть змінювати своє значення в процесі виконання програми. І аналогічно глобальним константам, до глобальних змінних можна звертатися використовувати їх будьде в файлі, в якому записується код програми. Бажано не зловживати використанням глобальних змінних, і якщо є можливість їх не використовувати в програмі, то краще від глобальних змінних відмовитися.

Прототипи функцій, які створюються розробником програми, повинні бути записані до моменту використання цих функцій в програмі (до моменту виклику функцій в певному фрагменті коду). Для невеликих проектів прототипи функцій можна записувати на початку програми після директив препроцесора. Якщо в програмі не передбачається використання функцій, які розробляються програмістом, то прототипи функцій будуть відсутні.

Функція `main()` є обов'язковим елементом в програмі на мові С. Ця функція виконує запуск виконання програми. В описі функції `main()` можуть міститися виклики інших функцій — як стандартної бібліотеки, так і функції сторонніх бібліотек, — при цьому необхідно підключити в програму відповідні заголовочні файли, також функція `main()` може містити виклики функцій, які створюються і описуються розробником програми в тексті програми.

Зазвичай, після опису функції `main()` записуються описи функцій, які створюються в даній програмі розробником програми.

Після написання тексту програми, для того, щоб отримати файл, призначений для виконання (файл з розширенням *.exe), необхідно виконати компіляцію коду програми. Загалом, етапи обробки коду програми для отримання .exe-файлу, показано на рис. 3.2 [3, 7].

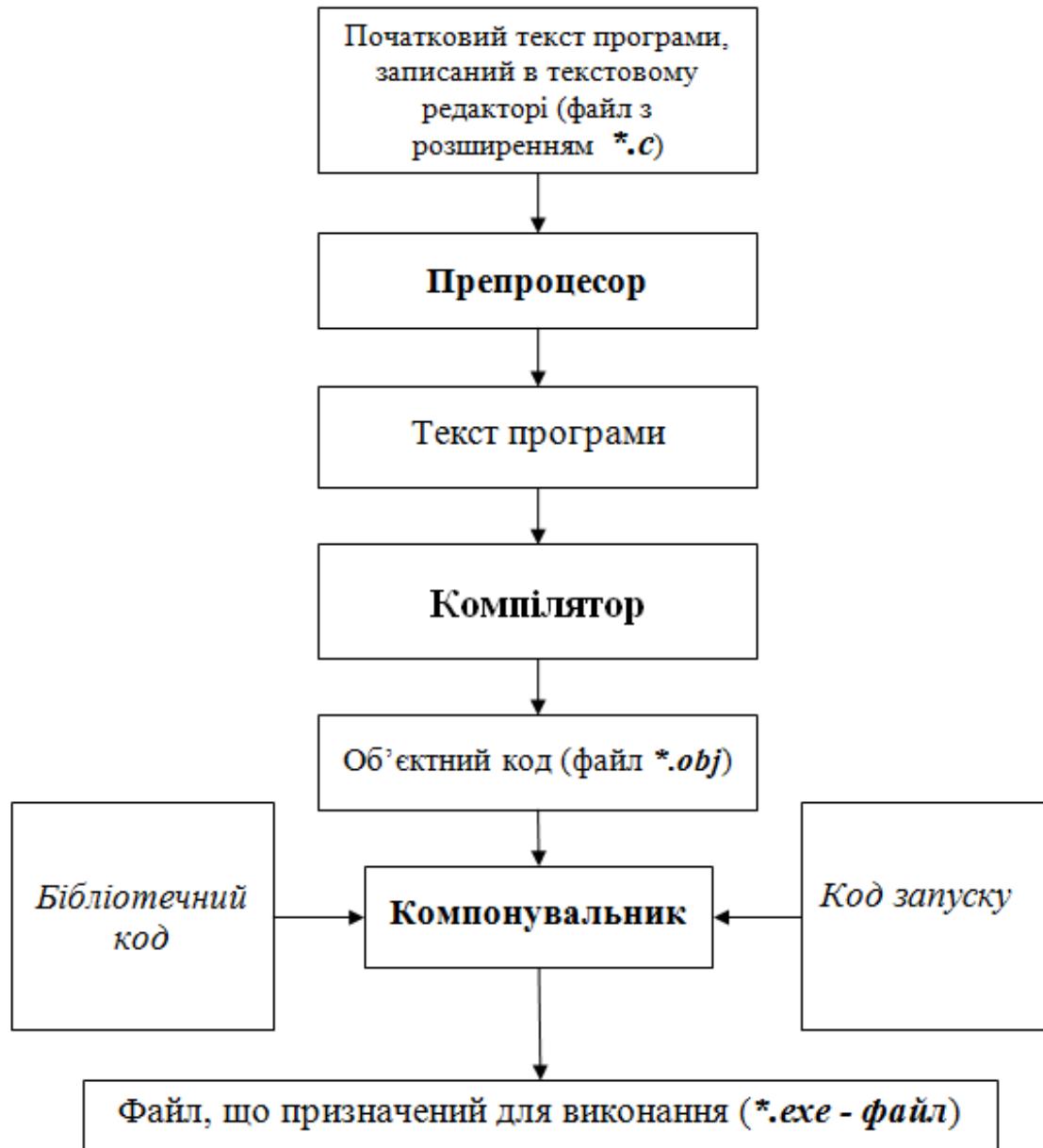


Рис. 3.2 — Етапи обробки тексту програми для отримання файлу, що призначений для виконання

3.2 Типи даних мови програмування С

При реалізації різноманітних алгоритмів виникає необхідність в оперуванні різного роду інформації. Ця інформація може зберігатися в константах, змінних, масивах тощо. Для правильного збереження та правильної інтерпретації відповідної інформації необхідно обрати відповідний тип даних для змінної, константи, масиву тощо. В мові програмування С існують типи даних, які призначені для зберігання цілих числових значень, а також для зберігання дійсних числових значень (тобто таких значень, які містять дробову частину). Цілі типи даних поділяються на **знакові цілі** типи та **беззнакові цілі** типи. Значення **беззнакових** цілих типів даних, а також **додатні** значення знакових цілих типів даних зберігаються в пам'яті комп'ютера в прямому коді, а **від'ємні** значення знакових цілих типів зберігаються в пам'яті комп'ютера в доповняльному коді.

Для збереження значень цілих типів використовується формат з фіксованою комою, при якому вважається, що десяткова кома розміщується за межами розрядної сітки так що розряди для зберігання дробової частини не передбачені. Таким чином, коли змінній цілого типу даних присвоюється значення, яке містить цілу та дробову частини, то дробова частина відсікається і не зберігається.

Для збереження значень дійсних типів даних, тобто типів даних які призначені для зберігання дійсних чисел, які містять цілу та дробову частини, використовується формат із плаваючою комою.

Для відповідного типу даних виділено відповідну кількість *біт* (або *байт*), що впливає на об'єм пам'яті, який виділяється на збереження значення обраного типу даних. Крім того, кількість бітів, яка виділена для відповідного типу даних, визначає діапазон значень, які можуть зберігатися змінною або константою обраного типу даних. В табл. 3.1 наведено перелік базових цілих типів даних — вказано назва типу даних, кількість бітів, які виділені для

збереження значень відповідного типу, а також вказано діапазон значень. Беззнакові типи даних в своїй назві містять службове слово **unsigned** [5, 7].

Табл. 3.1 — Базові цілі типи даних

<i>Тип даних</i>	<i>Розмір, біт</i>	<i>Діапазон значень</i>
unsigned char	1	0...255
char	1	-128...127
unsigned short	16	0...65535
short	16	-32768...32767
unsigned int	32	0...4 294 967 295
int	32	-2 147 483 368 ... 2 147 483 367
unsigned long	32	0...4 294 967 295
long	32	-2 147 483 368 ... 2 147 483 367

Розмір в бітах (або в байтах), а також діапазон можливих значень, для деяких типів даних може залежати від відповідної системи, і відрізнятися від показників наведених в табл. 3.1. Крім того, при перегляді стандарту мови C, можуть вводитися нові типи даних. Також на основі базових типів даних, можуть визначитися інші типи даних.

Знакові типи даних — це **char**, **short**, **int**, **long** [3—8]. Для того, щоб в явному вигляді вказати, що це знаковий тип даних, то може бути використане службове слово **signed**. Таким чином, назви знакових типів даних при написані тексту програм, можна записувати так:

```
signed char
signed short
signed int
signed long
```

Крім того, знакові типи `short` та `long` можуть в своїй назві містити службове слово `int`. А тип даних `int` може містити лише слово `signed`. Варіанти імен для відповідних типів даних вказано нижче.

Ім'я знакового цілого типу даних `long` може записуватися за допомогою будь-якого із наступних варіантів запису:

```
signed long int  
signed long  
long int  
long
```

Для запису імені знакового цілого типу даних `int` може використовуватися будь-який із наступних варіантів запису:

```
signed int  
signed  
int
```

Ім'я знакового цілого типу даних `short` може записуватися за допомогою будь-якого із наступних варіантів запису:

```
signed short int  
signed short  
short int  
short
```

Ім'я знакового цілого типу даних `char` може записуватися за допомогою будь-якого із наступних варіантів запису:

```
signed char  
char
```

Таким чином, для кожного типу даних доступні декілька варіантів запису імені відповідного типу. Аналогічно, і *беззнакові* цілі типи можуть мати різні варіанти написання імені відповідного типу [3—8]. Наприклад, ім'я беззнакового цілого типу `unsigned long` може записуватися так:

```
unsigned long int  
unsigned long
```

Ім'я беззнакового цілого типу **`unsigned int`** може записуватися так:

```
unsigned int  
unsigned
```

Ім'я беззнакового цілого типу **`unsigned short`** може записуватися так:

```
unsigned short int  
unsigned short
```

Для беззнакового цілого типу **`unsigned char`** доступний тільки один варіант написання імені цього типу даних:

```
unsigned char
```

Інформація про **максимальне** та **мінімальне** значення, яке може бути збережене в змінній або в константі відповідного цілого типу даних, зберігається в заголовочному файлі **limits.h**.

Для зберігання значень дійсних чисел, які мають дробову частину, використовуються типи даних, які приведені в табл.3.2 [7, 8].

Табл. 3.2 — Дійсні типи даних

Тип даних	Розмір, біт	Діапазон значень
float	32	Мінімальне додатне значення: 1.75494×10^{-38} Максимальне додатне значення: $3.402823 \times 10^{+38}$
double	64	Мінімальне додатне значення: $2.225074 \times 10^{-308}$ Максимальне додатне значення: $1.797693 \times 10^{+308}$
long double	В залежності від системи може займати 80 біт, 96 біт або 128 біт	

Для збереження значень типів даних **float**, **double** та **long double** використовується формат з плаваючою комою.

Інформація про знак числа для типів **float**, **double** та **long double** зберігається в найстаршому біті. Якщо число *від'ємне* — в найстаршому біті зберігається 1. Якщо число *додатне* — в найстаршому біті зберігається значення 0.

При написанні програми окрім вибору типів для збереження відповідних даних, якими необхідно оперувати в ході виконання програми, також необхідно правильним чином обирати імена для змінних, констант, функцій тощо, які оголошуються в програмі. В якості імені виступає *ідентифікатор*. При виборі імені (при складанні ідентифікатора) необхідно притримуватися відповідних вимог щодо правил формування ідентифікаторів. **Основні вимоги щодо назв ідентифікаторів наступні** [3, 4, 7]:

1. Ідентифікатор може формуватися із цифр, знаку підкреслення та літер англійського алфавіту, причому ідентифікатор не може починатися із цифри та не може містити пробіли.
2. Ідентифікатори чутливі до регістру символу. Таким чином, одне й те ж саме слово, яке записано великими літерами та маленькими літерами, наприклад, START, Start та start — це три різні ідентифікатори (тобто три різні, не співпадаючі між собою, імені).
3. Ідентифікатори не повинні співпадати із службовими словами. До службових слів (або ще їх називають ключовими словами) відносять назви типів даних, назви операторів мови С та ін.
4. Рекомендується для ідентифікаторів обирати осмислені назви, тобто такі, щоб в назві ідентифікатора було зашифровано призначення відповідної змінної, константи чи функції, для якої обирається ідентифікатор.
5. Рекомендується ідентифікатори обирати коротшими ніж 32 символи (в деяких системах символи в назвах ідентифікаторів починаючи з 32 — ігноруються).

6. Рекомендується не використовувати символ підкреслення в якості початкового символу в назві ідентифікатора.

В наступному прикладі приводиться програмний код простої програми, в якій за допомогою директиви препроцесора `#define` створюються дві константи `N` та `SIZE` цілого типу даних (по-замовчуванню, це тип даних `int`). А в функції `main()` оголошується змінна `root` типу `float`, а також ініціалізуються змінні `coefficient_1` та `factor` також типу `float`. Відмінність між оголошенням змінної та ініціалізацією змінної полягає в тому, що при ініціалізації змінній одразу присвоюється певне значення. Змінні `root`, `coefficient_1` та `factor` — це локальні змінні. Для констант та змінних були обрано імена — `N`, `SIZE`, `root`, `coefficient_1`, `factor`, — це і є ідентифікатори (які повинні відповідати вище зазначенним вимогам). На відмінну від глобальних змінних, які видимі всюди у файлі, локальні змінні мають область видимості тільки той блок, де вони оголошуються. В даному випадку змінні оголошені в межах функції `main()`, тому ці змінні можуть використовуватися в межах цієї функції. Опис функції `main()` (або іншими словами — тіло функції) записується між відкриваючою та закриваючою фігурними дужками. Обов'язковим оператором, яким функція `main()` завершується, є оператор `return 0;` як показано нижче:

```
#define SIZE 10
#define N 15

int main()
{
    float root;
    float coefficient_1 = 10.0;
    float factor = 2.5;

    root = coefficient_1 / factor;

    return 0;
}
```

В наведеному прикладі, відсутні заголовочні файли, оскільки не викликаються ніякі функції стандартної бібліотеки. В програмі лише задаються константи та змінні, виконується операція ділення `coefficient_1/factor` і

результат присвоюється змінній `root`. В наведеному прикладі значення змінних `coefficient_1` та `factor` задаються на етапі їх оголошення. А початкове значення для змінної `root` не вказується. В такому випадку `root` є невизначеню, тобто невідомо чому вона дорівнює. Потрібно пам'ятати, що локальні змінні, які оголошуються, але не ініціалізуються, залишаються невизначеними. Шляхом виконання оператора `root = coefficient_1 / factor;` змінній `root` буде присвоюватися результат, що отриманий в ході ділення значення змінної `coefficient_1` на значення змінної `factor`. В представленаому фрагменті коду оголошення змінної `root`, а також ініціалізація змінних `coefficient_1` та `factor` записуються на окремих рядках. Враховуючи, що ці змінні мають один тип даних, тому відповідні дії можна було б записати в один рядок, і тоді фрагмент коду можна переписати таким чином:

```
#define SIZE 10
#define N 15

int main()
{
    float root, coefficient_1 = 10.0, factor = 2.5;

    root = coefficient_1 / factor;

    return 0;
}
```

Однак рекомендується не змішувати в одному рядку оголошення змінних та ініціалізацію змінних. Представлений вище фрагмент коду демонструє поганий стиль написання програми. Тому краще його переписати таким чином:

```
#define SIZE 10
#define N 15

int main()
{
    float root;
    float coefficient_1 = 10.0, factor = 2.5;

    root = coefficient_1 / factor;

    return 0;
}
```

Загалом, процедура оголошення змінної може бути представлена в такому форматі [7]:

тип_даних [пробіл] ім'я_змінної;

Наприклад:

```
int x1, start_point;  
unsigned char Status;  
double result, offset;
```

В наведених фрагментах коду змінні `x1` та `start_point` — це змінні знакового цілого типу даних `int`. Змінна `Status` — це змінна беззнакового типу даних `unsigned char`. Змінні з іменами `result` та `offset` — це змінні дійсного типу даних `double`. Змінні в даному прикладі оголошуються, але не ініціалізуються, тобто їм не надається одразу певне значення в момент їх оголошення.

Процедура ініціалізації змінної дуже схожа на оголошення за виключенням того, що змінній одразу присвоюється певне значення. В загальному випадку ініціалізація змінної може бути записана в такому форматі :

тип_даних [пробіл] ім'я_змінної = значення_змінної;

Наприклад:

```
int x1=5, start_point=-10;  
unsigned char Status=2;  
double result=0.0, offset=-1.0;
```

Для поліпшення візуального сприйняття тексту програми, зліва і справа від знаку дорівнює можна використати пробільні символи:

```
int x1 = 5, start_point = -10;  
unsigned char Status = 2;  
double result = 0.0, offset = -1.0;
```

Кожний запис по ініціалізації (або оголошенню) змінних, які мають одинаковий тип даних і можуть записуватися в одному рядку, можна записувати на кожному окремому рядку. При цьому в кінці рядка необхідно обов'язково записувати символ «крапка з комою»:

```
int x1 = 5;
int start_point = -10;
unsigned char Status = 2;
double result = 0.0;
double offset = -1.0;
```

Схожим чином можуть ініціалізовуватися константи відповідного типу. Таким чином, крім створення констант за допомогою директиви `#define` можна також записувати в програмі константи відповідно до наступного формату запису:

const [пробіл] тип_даніх [пробіл] ім'я_константи = значення_константи;

На відміну від ініціалізації змінної, при ініціалізації константи перед типом даних необхідно записувати службове слово `const`.

В наведеному прикладі виконується ініціалізація констант, оголошення змінних та ініціалізація змінної:

```
const float pi = 3.14159;
const unsigned int scale = 5;
unsigned int amount;
int sum = 0;
float output;
```

В даному прикладі константа `pi` типу даних `float` отримує значення `3.14159`. Константа `scale` типу даних `unsigned int` отримує значення `5`. Змінна `amount` типу даних `unsigned int` оголошується (але не ініціалізується, тобто змінній не присвоюється ніяке значення на етапі оголошення, а якщо змінна `amount` є локальною змінною, то значення її – невідоме, тобто змінна не визначена). Змінна `sum` типу даних `int` ініціалізується (zmінній присвоюється

значення 0). Змінна `output` типу даних `float` оголошується (якщо змінна `output` — це локальна змінна, то значення її – невідоме, тобто змінна не визначена).

Враховуючи, що при написанні тексту програми, важливо акуратно форматувати текст, щоб легше було його сприймати, аналізувати, виправляти в ньому помилки та вносити зміни в разі потреби, тому необхідно притримуватися певних вимог до оформлення тексту. Ці вимоги можуть визначатися всередині групи розробників, які працюють над певним проектом, або можна слідувати загальноприйнятим нормам. Форматувати текст можна за допомогою пробільних символів — символу «пробіл» або «табуляція». Наприклад, попередній фрагмент коду міг би бути переписаний таким чином для більш зручного візуального сприйняття:

```
const float          pi      = 3.14159;
const unsigned int  scale   = 5;
unsigned int         amount;
int                 sum     = 0;
float               output;
```

Мова програмування С дозволяє гнучко підходити до форматування тексту програми. Тому важливо використовувати всі надані нею можливості для якісного написання тексту програми.

Крім форматування тексту, для поліпшення читабельності коду програми необхідно застосовувати коментарі, які, наприклад, пояснюють призначення тих або інших змінних, констант чи функцій, також коментарі можуть містити опис дій, які передбачаються до виконання в програмі [3, 6, 7].

В мові програмування С доступні два види коментарів. Перший вид коментарів — це однорядкові коментарі. Для того, щоб записати однорядковий коментар, — необхідно поставити символи `//` («скісна риска», ще називають цей символ «прямий слеш») і потім записувати текст коментаря.

Другий вид коментарів — це багаторядкові коментарі. Початок коментаря позначається символами `/*` («скісна риска» та «зірочка»), а завершення коментаря позначається символами `*/` («зірочка» та «скісна риска»).

Коментарі — не компілюються. Тому, крім використання коментарів в якості додаткового пояснення окремих фрагментів коду, коментарі також можуть застосовуватися щоб, наприклад, при компіляції не компілювати відповідний фрагмент коду. Тому, щоб залишити фрагмент коду в програмі, а не видаляти його перед компіляцією, його можна закоментувати.

Приклад використання різних типів коментарів:

```
/*
Це багаторядковий коментар, який може використовуватися
для пояснення дій, які виконуються в програмі.
При компіляції програми — текст, який записаний
в коментарях не компілюється
*/
// Це однорядковий коментар. Текст коментаря не компілюється

#define SIZE 10
#define N 15

// Далі буде записуватися тіло функції main()
int main()
{
    const float pi = 3.14159;           // константа, яка зберігає число пі
    const unsigned int scale = 5;       // деякий константний коефіцієнт
    unsigned int amount;               // змінна має тип даних unsigned int
    int sum = 0;                      // змінна, що зберігає обрахунок суми
    float output;                    // змінна для зберігання результату

    return 0;
}
```

За допомогою директиви препроцесора `#define` можна створювати константи яким буде надаватися значення деякого одиночного символу або символьного рядка. В наведеному нижче прикладі константа `SingleSymbol` призначена для зберігання символу `*` («зірочка»), константа `AcademicDiscipline` призначена для зберігання такого символьного рядка: `Informatics Programming Fundamentals`. Крім цього створюється константа `E` для зберігання дійсного числа `2.71`. Нижче приводиться приклад:

```
#define SingleSymbol '*'
#define AcademicDiscipline "Informatics. Programming Fundamentals"
#define E 2.71
```

При створенні константи, яка призначена для зберігання одиночного символу, відповідний символ повинен записуватися в одинарних лапках (в наведеному прикладі символ «зірочка» записується в одинарних лапках). На відміну від одиночного символу, символьний рядок повинен записуватися в подвійних лапках.

Порівнюючи запис для оголошення змінних та запис, в яких створюються константи за допомогою директиви `#define`, можна помітити, що в записі константи за допомогою директиви `#define` не вказується тип даних. Крім того, в кінці рядка не ставиться символ «крапка з комою».

Особливість використання директиви `#define` полягає в тому, що всюди в тексті програми (окрім коментарів), де будуть зустрічатися ідентифікатори `SingleSymbol`, `AcademicDiscipline` та `E`, перед тим як програма піде на компіляцію, препроцесор поставить замість вказаних ідентифікаторів відповідні значення, які стоять поряд з ними в директиві `#define`. Таким чином, замість імені константи `SingleSymbol` у відповідному фрагменті програми буде поставлений символ `*`, замість імені константи `AcademicDiscipline` у відповідному фрагменті коду буде вставлено рядок `Informatics. Programming Fundamentals`, а замість імені константи `E` буде вставлено значення `2.71`. Таким чином, відбудеться підстановка відповідних значень замість імен констант, де ці константи використовуються. Після цього код програми, який був видозмінений зазначеним чином, буде оброблятися компілятором. Тому, наприклад, друкарська помилка (опечатка) при записі значення константи може привести до того, що неправильне значення буде підставлено у відповідний фрагмент коду програми під час обробки тексту програми препроцесором, що може стати причиною появи помилок при компіляції програми і неможливості отримати вихідний ехе-файл.

В наведених вище фрагментах програмного коду не відбувалося введення значень з клавіатури, а також виведення значень на екран, не виконувалися виклики функцій стандартної бібліотеки. Для того щоб розширити функціональні можливості програми, наприклад, шляхом створення деякої інтерактивної взаємодії із користувачем, необхідно використовувати функції по зчитуванню даних з клавіатури, щоб користувач мав можливість вводити вхідні дані, а також забезпечити можливість відображення результату на екрані монітора, наприклад, в консольному вікні, щоб користувач зміг переглянути результати виконання програми. Існує ряд функцій стандартної бібліотеки, які призначенні для виконання вказаних дій. Зокрема, для зчитування вхідних даних з клавіатури, використовується функція `scanf()`. Для виведення даних на екран монітору в консольне вікно — використовується функція `printf()`. Для того, щоб мати можливість використовувати ці функції в програмі, необхідно за допомогою директиви препроцесора `#include` підключити заголовочний файл `stdio.h`.

В загальному випадку, функція `printf()` має наступний формат запису [7]:

```
printf("рядок формату" [, аргумент1, аргумент2, ..., аргументN] );
```

В наведеному прикладі елементи, які записані в квадратних дужках, можуть бути відсутні в конкретному варіанті запису виклику функції `printf()`.

В наведеному прикладі `аргумент1, аргумент2, ..., аргументN` — це перелік аргументів (перелік параметрів), які передаються у функцію `printf()` для виведення значень даних аргументів на екран монітора. В якості аргументів можуть виступати: змінні; константи; арифметичні чи логічні вираз, значення яких обчислюються і виводяться на екран; значення результату, який повертається деякою функцією, виклик якої записується на позиції аргументу у функції `printf()` та ін. **Рядок формату** — це певний запис, який може містити деякий текст для відображення на екрані монітору, а також у випадку наявності аргументів, які необхідно вивести в консольне вікно **рядок формату** повинен містити *спеціальні символи*, які дозволяють коректно відобразити значення

відповідних аргументів, причому значення аргументів будуть розміщуватися на позиції відповідного *спеціального символу* в рядку формату.

Функція `printf()` може містити тільки **рядок формату** без будь-яких аргументів, які слідують за ним, в такому випадку **рядок формату** може містити деякий текст для відображення в консольному вікні.

Загалом **рядок формату** крім текстової інформації, у випадку наявності у виклику функції `printf()` переліку параметрів, буде містити набір *спеціальних символів*, які призначені для форматування виводу значень параметрів, а також для коректного відображення цих значень. Набір таких *спеціальних символів* починається з символу `%`, і в загальному випадку такий запис називається **специфікатором формату**.

Специфікатор формату має таку структуру [3, 7]:

`%[Прапор][Ширина].[Точність]РозмірТип`

В представленій структурі специфікатора формату елементи, які записані в квадратних дужках, є необов'язковими і можуть не використовуватися при виклику функції `printf()`. Лише поле **Тип** є обов'язковим і визначає тип перетворення відповідного аргументу із переліку аргументів у виклику функції `printf()`, тим самим впливаючи на відображення відповідного значення аргументу в консольному вікні. Поле **Тип** визначає яким чином виконувати інтерпретацію відповідного аргументу функції `printf()`. В табл. 3.3 приводяться специфікатори типів та надано короткий коментар по кожному із них [3].

Табл. 3.3 — Специфікатори типу

Специфікатор Тип	Призначення
<code>%c</code>	Виведення одиночного символу
<code>%d</code>	Виведення десяткового знакового цілого числа
<code>%e</code>	Виведення числа з плаваючою комою (плаваючою точкою). Значення виводиться в експоненціальній формі із символом експоненти « <code>e</code> » в нижньому регістрі.

Продовження табл. 3.3

Специфікатор Тип	Призначення
%E	Виведення числа з плаваючою комою. Значення виводиться в експоненціальній формі із символом експоненти «E» у верхньому регистрі.
%f	Виведення числа з плаваючою комою. Записується в десятковій формі.
%g	Використовується специфікатор %f або %e, в залежності від числа, що потребує виведення. Форма %e використовується якщо показник степеня менше -4 або більший чи рівний заданої точності числа
%G	Використовується специфікатор %f або %E, в залежності від числа, що потребує виведення. Форма %E використовується якщо показник степеня менше -4 або більший чи рівний заданої точності числа
%i	Виведення десяткового знакового цілого числа (аналогічно %d)
%o	Виведення беззнакового цілого числа у вісімковій системі числення
%p	Виведення вказівника
%s	Виведення символьного рядка
%u	Виведення десяткового беззнакового цілого числа
%x	Виведення беззнакового цілого числа в шістнадцятковій системі числення. Шістнадцяткові символи записуються в нижньому регистрі
%X	Виведення беззнакового цілого числа в шістнадцятковій системі числення. Шістнадцяткові символи записуються в верхньому регистрі
%%	Виведення знаку процент %

Крім специфікаторів типів, в рядку формату у функції `printf()` можуть розміщуватися **escape-послідовності**, які дозволяють управляти виведенням текстової інформації, наприклад, переводити курсор виведення в консольному вікні на наступний рядок або робити відступ (табуляцію) між окремими частинами текстової інформації, або переводити курсор на початок поточного рядка тощо.

Escape-послідовності починаються із зворотної скісної риски (або по-іншому ще називають — зворотний слеш). Перелік деяких escape-послідовностей приведено в табл. 3.4 [5, 6].

Табл. 3.4 — Escape-послідовності

Escape-послідовність	Призначення
\a	Звуковий сигнал динаміку комп’ютера
\b	Переведення курсору на одну позиції вліво
\n	Переведення курсору на новий рядок
\r	Переведення курсору на початок поточного рядка
\t	Табуляція
\'	Символ «апостроф»
\"	Символ «подвійні лапки»
\\\	Символ зворотної скісної риски (зворотний слеш)
\?	Символ «знак питання»
\0	Нульовий символ

Нижче наведено приклади використання escape-послідовності \n для запису відповідної текстової інформації з нового рядка в консольному вікні, також показано використання відповідних специфікаторів формату для виведення в консольне вікно: одиночного символу (%c); символьного рядка (%s); десяткового знакового числа (%d), десяткового беззнакового числа (%u); виведення беззнакового цілого числа в вісімковій системі числення (%o); виведення беззнакового цілого числа в шістнадцятковій системі числення (%X); виведення дійсного числа в десятковій формі (%f); виведення дійсного числа в експоненціальній формі (%E):

```

#include<stdio.h>

#define SingleSymbol '*'
#define AcademicDiscipline "Informatics. Programming Fundamentals"

int main(void)
{
    int x = -24;
    unsigned int y = 207;
    float t = -10.987654321;
    char ch = '!';           // символ «знак оклику»
    char str[] = "Algorithms"; // символьний рядок

    // в консольному вікні тричі ставиться символ табуляція
    // виводиться текст Output:
    // і курсор переводиться на новий рядок за допомогою \n

    printf("\t\t\tOutput:\n");

    // За допомогою \n курсор переводиться на новий рядок
    // З нового рядка виводиться текст Symbols:
    // і через пробіл виводяться одиночні символи.
    // Перший символ визначається константою SingleSymbol
    // Другий символ – знак «плюс»
    // Третій символ – зберігається в змінній, що має
    // ім'я ch та тип даних char (це символ «знак оклику»)

    printf("\nSymbols: %c %c %c", SingleSymbol, '+', ch);

    // За допомогою \n\n курсор двічі переводиться на новий рядок
    // З нового рядка виводиться текст Subject:
    // і через пробіл виводяться символьні рядки.
    // Перший символьний рядок визначається константою AcademicDiscipline
    // Другий символьний рядок – це рядок "and" що записаний
    // на другій позиції в переліку аргументів функції printf()
    // Текст третього символьного рядка – визначається
    // символьним рядком str

    printf("\n\nSubject: %s %s %s", AcademicDiscipline, "and", str );

    // За допомогою \n\n курсор двічі переводиться на новий рядок
    // З нового рядка виводиться текст integer number =
    // і виводиться значення змінної x. Числова величина
    // відображається на місці розміщення
    // специфікатора формату %d

    printf("\n\ninteger number = %d", x);
}

```

```

// За допомогою \n\n курсор двічі переводиться на новий рядок
// З нового рядка виводиться текст unsigned integer number =
// і виводиться значення змінної у. Причому значення змінної у
// відображається як десяткове число, а також як вісімковий код,
// і як шістнадцятковий код.
// Перший специфікатор формату %u – узгоджується з першим параметром,
// в якості якого виступає змінна на ім'я у
// Другий специфікатор формату %o – узгоджується з другим параметром,
// в якості якого також виступає змінна на ім'я у
// Третій специфікатор формату %X – узгоджується з третім параметром,
// в якості якого знову виступає змінна на ім'я у

printf("\n\nunsigned integer number: %u  %o  %X", y, y, y);

// За допомогою \n\n курсор двічі переводиться на новий рядок
// З нового рядка виводиться текст float number:
// після текстового повідомлення відображається значення
// змінної t в десятковій формі запису, потім відображається
// пробіл, табуляція, пробіл, і відображається значення
// змінної t в експоненціальній формі запису.
// Першому специфікатору формату %f відповідає перший
// параметр ( це змінна на ім'я t )
// Другому специфікатору формату %E відповідає другий
// параметр ( це змінна на ім'я t, яка записана на другій позиції )
printf("\n\nfloat number: %f \t %E", t, t);

printf("\n"); // переведення курсора на новий рядок
return 0;
}

```

Результати виконання наведеної програми по виведенню значень різних типів даних в консольне вікно показані на рис. 3.3.

```

Output:
Symbols: * + !
Subject: Informatics. Programming Fundamentals and Algorithms
integer number = -24
unsigned integer number: 207 317 CF
float number: -10.987655      -1.098765E+001

```

Рис. 3.3 — Виведення значень різних типів за допомогою функції printf()

Крім специфікатора **Тип** (перелік специфікаторів приведені в табл. 3.3), також може використовуватися модифікатор **Розмір**.

Модифікатор **Розмір** застосовується у функції printf() з метою надання інформації щодо розміру даних, які передаються у функцію printf() в якості параметрів. Враховуючи, що, наприклад, цілі типи даних short, int, long, або

дійсні типи даних `float`, `double` чи `long double` займають в пам'яті різну кількість *Байт*, тобто мають різний розмір, то при роботі з деякими типами даних необхідно крім специфікатора **Тип** також вказувати модифікатор **Розмір** для коректного відображення даних в консольному вікні. Модифікатор **Розмір** застосовується разом із специфікатором **Тип** (табл. 3.5) [3].

Табл. 3.5 — Модифікатор Розмір

Модифікатор Розмір	Призначення
<code>h</code>	Використовується разом із специфікатором цілих типів (<code>%d</code> <code>%i</code> <code>o%</code> <code>%u</code> <code>%x</code> <code>%X</code>) для типу даних <code>short</code> та типу даних <code>unsigned short</code>
<code>l</code>	Використовується разом із специфікатором цілих типів (<code>%d</code> <code>%i</code> <code>o%</code> <code>%u</code> <code>%x</code> <code>%X</code>) для типу даних <code>long</code> та типу даних <code>unsigned long</code>
<code>L</code>	Використовується разом із специфікатором дійних типів (<code>%e</code> <code>%E</code> <code>%f</code> <code>%g</code> <code>%G</code>) для типу даних <code>long double</code>

Для впливу на відображення даних в консольному вікні використовується модифікатор **Точність** (табл. 3.6).

Модифікатор **Точність** може бути записано в двох варіантах — перший варіант: записується символ «точка» та деяке число, тобто формат запису такий:

.число

В парі із відповідним специфікатором типу, який застосовується для відображення в консольному вікні значень відповідного типу даних, дія модифікатора **Точність** буде визначатися в залежності від специфікатора типу.

Другий варіант запису модифікатора **Точність** передбачає використання символу «зірочка», в цьому випадку формат запису такий:

.*

Табл. 3.6 — Модифікатор Точність

Модифікатор Точність	Призначення
.число	<p>1. Разом із специфікатором цілих типів (%d %i 0% %u %x %X) значення модифікатора Точність визначає мінімальну кількість цифр, які відображаються на екрані при виведенні параметра. Якщо кількість цифр в значенні параметра, що виводиться в консольне вікно, менша ніж вказане значення Точності, тоді зліва від значення, яке відображається в консольному вікні, додається відповідна кількість нулів, щоб сукупна кількість цифр дорівнювала значенню Точності. Якщо кількість цифр в параметрі, що виводиться в консольне вікно, більша ніж значення Точності, тоді значення точності ігнорується.</p> <p>2. Разом із специфікатором %e або %E або %f визначає кількість символів дробової частини, що виводиться після десяткової коми (десяткової точки). По замовчуванню значення модифікатора Точність дорівнює 6. Якщо значення модифікатора Точність задати рівним 0 або після символу «точка» не записати число, тоді десяткова кома (десяткова точка) не відображається в значенні результату, що виводиться в консольне вікно.</p> <p>3. Разом із специфікатором %g або %G визначає максимальну кількість значущих цифр.</p> <p>4. Разом із специфікатором %s визначає максимальну кількість символів, що виводяться в консольне вікно. Якщо символний рядок коротший за вказану Точність, то він виводиться повністю. Якщо символний рядок довший за вказану Точність, то виводяться лише символи починаючи з першого і до символу, номер позиції якого в символному рядку, дорівнює вказаній Точності.</p>

Модифікатор Точність	Призначення
.*	<p>Такий варіант запису модифікатора Точність працює аналогічно переліченим вище, за виключенням того, що саме значення Точності задається у функції printf() як окремий числовий параметр, що записується перед параметром, який виводиться в консольне вікно.</p> <p>Наприклад:</p> <pre data-bbox="441 781 1029 815">printf("%.*s", 4, "123456");</pre> <p>Точність задається у вигляді параметра 4. Застосовується разом із специфікатором %s. Таким чином, на екрані будуть відображатися перші 4 символи того символьного рядка, який виступає в якості параметра функції printf(), в наведеному прикладі символьний рядок має вигляд: 123456</p> <p>Тому на екрані в консольному вікні буде відображатися наступне:</p> <p style="color: red;">1234</p>

Модифікатор **Ширина** визначає мінімальну ширину того поля, куди виводиться значення параметру. Значення модифікатора **Ширина** — так само як і значення модифікатора **Точність** — це невід'ємне десяткове ціле значення.

Так само як і для модифікатора **Точність**, модифікатор **Ширина** може записуватися в двох варіантах (табл. 3.7). Перший варіант — вказується число разом із записом відповідного специфікатора типу. Другий варіант — записується символ «зірочки», і значення модифікатора **Ширина** виступає в якості параметра функції printf() і записується в переліку параметрів перед значенням відповідного параметру (а в разі якщо значення модифікатора **Точність** задається у вигляді параметру, що записується в переліку параметрів у функції printf()), — параметр модифікатора **Ширина** записується перед

параметром **Точність** і параметром, до якого відповідні модифікатори застосовуються)

Табл. 3.7 — Модифікатор Ширина

Модифікатор Ширина	Призначення																				
число	Разом із відповідним специфікатором визначає мінімальну ширину поля, в яке виводиться результат в консольному вікну. Якщо заданої ширини недостатньо для відображення значення на екрані, то результат не усікається, а виводиться в поле необхідної ширини.																				
*	<p>Такий варіант запису модифікатора Ширина працює аналогічно варіанту вище, за виключенням того, що саме значення Ширини задається у функції printf() як окремий параметр. Наприклад:</p> <pre>printf("%.*f", 10, 3, 95.123456789);</pre> <p>В даному випадку Ширина та Точність задаються як параметри, що дорівнюють 10 і 3. Таким чином, умовне поле в яке буде виводитись результат можна представити в такому вигляді:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="width: 10px;"></td> </tr> <tr> <td>9</td> <td>5</td> <td>.</td> <td>1</td> <td>2</td> <td>3</td> <td></td> <td></td> <td></td> <td></td> </tr> </table> <p>Якщо Ширина буде вказана малою, недостатньою для відображення значення,— поле буде розширене до необхідної ширини.</p>											9	5	.	1	2	3				
9	5	.	1	2	3																

Модифікатор **Пропор** використовується для впливу на характер відображення значення в консольному вікні при виведенні. Наприклад, впливає на вирівнювання результату, що відображається в умовному полі в консольному вікні, дозволяє записувати знак «+» при виведенні додатних чисел тощо. Перелік позначень модифікатора **Пропор** та результат, який отримується

при застосуванні відповідного позначення для модифікатора **Прапор**, приведено в табл. 3.8 [3].

Табл. 3.8 — Модифікатор Прапор

Модифікатор Прапор	Призначення
—	Вирівнювання результату, що виводиться в консольне вікно, по лівому краю поля в рамках заданої ширини. (По замовчуванню, без використання даного Прапору, вирівнювання відбувається по правову краю поля)
+	Відображається знак «+» для доданих значень, та знак «—» для від'ємних значень, якщо виводиться значення знакового типу даних. (По замовчуванню, без використання даного Прапору, при виведенні додатних значень знак «+» не відображається)
пробіл	Якщо виводиться додатне значення, то знак «+» не відображається, але ставиться символ «пробіл» на його місці. Якщо виводиться від'ємне значення, ставиться знак «—». Дане значення Прапору ігнорується, якщо разом з цим Прапором записується Прапор «+».
0	Заповнює поле початковими нульовими символами щоб поле із заданою Шириною було заповнене повністю. Якщо Прапор «0» використовується разом із «—», тоді Прапор «0» ігнорується.
#	Виводиться нуль в старшому розряді при використанні із специфікатором %0 Виводиться 0x або 0X при використанні із специфікаторами %x та %X При використанні із специфікаторами для дійсних типів %e або %E , або %f забезпечує виведення десяткової коми (десяткової точки).

Для зчитування даних з клавіатури використовується функція **scanf()**.

Функція **scanf()** аналогічно функції **printf()** також повинна записуватися із відповідним специфікатором формату для коректної інтерпретації та збереження відповідних вхідних даних.

При роботі із функцією **scanf()** можуть використовуватися специфікатори, які приведені в табл.3.9 [3, 7].

Табл. 3.9 — Специфікатори формату функції **scanf()**

Специфікатор Типу	Призначення
%c	Введення одиночного символу
%d	Введення цілого знакового десяткового числа
%e, %E %f, %g %G	Введення дійсних чисел із плаваючою комою (плаваючою точкою)
%i	Введення цілого знакового числа у вигляді десяткового числа або вісімкового чи шістнадцяткового коду
%o	Введення цілого знакового числа у вісімковій системі числення
%p	Введення значення вказівника
%s	Введення символного рядку. Введення починається з першого символу, який відрізняється від пробільного symbolu. Введення продовжується до першого пробільного symbolu.
%u	Введення цілого беззнакового десяткового числа
%x, %X	Введення цілого знакового числа в шістнадцятковій системі числення

Крім специфікаторів, також можуть застосовуватися відповідні модифікатори, при зчитуванні та збереженні значень з клавіатури, наприклад, у змінну відповідного типу.

Наприклад, для типу даних `double` використовується модифікатор **`l`** (літера англійського алфавіту *l*) разом із специфікатором для роботи з дійсними типами: `%e`, `%E` `%f`, `%g` `%G`.

Для типу даних `long double` використовується модифікатор **`L`** разом із специфікаторами для роботи з дійсними типами: `%e`, `%E` `%f`, `%g` `%G`.

При введенні значення з клавіатури у відповідну змінну за допомогою функції `scanf()`, варто притримуватися наступного формату [3, 7]:

```
scanf ("% [Модифікатор] Спеціфікатор", &ім'я_змінної );
```

Модифікатор є необов'язковим елементом, і для деяких типів даних він може не записуватися. Спеціфікатор формату є обов'язковим. При зчитуванні значення з клавіатури у відповідну змінну, біля імені змінної у функції `scanf()` обов'язково необхідно записувати знак `&` (цей знак називається «Амперсанд»). Важливо контролювати, щоб тип даних змінної та Спеціфікатор формату узгоджувалися.

Приклад використання функції `scanf()` по введенню значення змінної, що має ім'я `input` для якої обрано тип даних `float`, показано нижче:

```
float input;  
  
printf("input=");  
scanf("%f", &input);
```

В наведеному прикладі, перед тим як вводити значення з клавіатури і присвоювати його змінній `input`, в консольному вікні буде відображене запит для користувача `input=` та буде очікуватися введення числового значення. Важливо забезпечувати інтерактивну взаємодію із користувачем, щоб інформувати що саме очікується від користувача при виконанні програми.

Можна одним викликом функції `scanf()` забезпечувати введення одразу декількох значень. Нижче показано приклад застосування такого варіанту:

```
float input;  
int number;  
double factor;
```

```

printf("Ведіть числа типу float, int та double:");
scanf("%f%d%lf", &input, &number, &factor);

printf("\ninput=%f\nnumber=%d\nfactor=%f", input, number, factor);

```

Введення значень змінних `input`, `number` та `factor` може записуватися в один рядок, натискаючи пробіл після введення чергового значення (наприклад, ввели перше значення — дійсне число (яке буде зберігатися в змінній `input`), натиснули пробіл, і перешли до введення наступного значення, яке повинно бути цілим знаковим числом (що буде зберігатися в змінній `number`), після натискання пробілу необхідно ввести третє число, що буде зберігатися в змінній `factor`, завершення введення генерується натисканням клавішею `Enter`. Замість пробілу, після введення кожного числового значення, можна натискати клавішу `Enter`). Останній рядок коду, в якому викликається функція `printf()` призначений для виведення значень відповідних змінних в консольне вікно з метою перевірки правильності збереження введених даних.

Можна помітити, що для змінної на ім'я `factor` типу `double`, для введення значення з клавіатури, використовуючи для цього функцію `scanf()`, записується специфікатор `%lf`, а для виведення значення цієї змінної в консольне вікно за допомогою функції `printf()`, використовується спеціфікатор формату `%f`, який також використовується і для виведення змінної типу `float`.

Таким чином, для виведення значень типу `float` та типу `double` в функції `printf()` використовується спеціфікатор `%f`.

При введенні значень з клавіатури за допомогою функції `scanf()`, для типу `float` використовується спеціфікатор `%f`, а для типу `double` використовується спеціфікатор `%lf`.

Загалом, спеціфікатор `%lf` може використовуватися і для функції `printf()` при виведенні значень типу даних `double`:

```
printf("\ninput=%f\nnumber=%d\nfactor=%lf", input, number, factor);
```

Важливо пам'ятати, що для функції `scanf()` використання специфікатора `%f` при роботі із змінною типу `double` призведе до неправильного зчитування значень і збереження його в змінній, що в подальшому може привести до неочікуваних результатів в ході виконання програми.

3.3 Операції мови програмування С

В мові програмування С передбачено набір операцій, які використовуються для обробки даних. В табл. 3.10 представлена операції мови С у вигляді відповідних символів, що їх позначають [3, 5—8]. Операції різняться своїм рангом або, іншими словами, своїм пріоритетом. Найвищий пріоритет мають ті операції, які мають ранг 1. Найменший пріоритет має операція, у якої ранг 15. Операції одного рангу мають одинаковий пріоритет. Ранг (пріоритет) операції дозволяє визначати послідовність виконання операцій, якщо в одному виразі зустрічається більше ніж одна операція. Крім рангу, при виконанні операцій враховується правило асоціативності, яке вказує яким чином відбувається виконання операцій одного рангу — зліва направо чи справа наліво.

Ранг та асоціативність дозволяють правильним чином виконувати обрахунок складних виразів, в яких використовується певна кількість операцій, деякі із них можуть мати одинаковий ранг, а деякі — різні ранги.

Операції можуть бути також охарактеризовані кількістю операндів, або іншими словами, кількістю аргументів, яку потребує відповідна операція. Операції можна розділити на однооперандні операції (операція застосовується до одного аргументу) або по-іншому називають такі операції — унарні операції, двохоперандні операції (операція потребує двох аргументів) або по-іншому ще називають такі операції — бінарні операції, та трьохоперандна операція (операція передбачає використання трьох аргументів) або по-іншому таку операцію називають тернарна операція.

Табл. 3.10 — Операції мови С

Ранг операції	Позначення операції	Опис операції	Асоціативність
1	()	Виклик функції	Зліва направо
	[]	Індекс масиву	
	.	Звернення до елементу структури	
	->	Непряме звернення до елементу структури	
2	!	Логічне НЕ (логічне заперечення)	Справа наліво
	~	Побітова інверсія	
	++	Інкремент (збільшення на 1)	
	--	Декремент (зменшення на 1)	
	+	Унарний «плюс»	
	-	Унарний «мінус»	
	*	Розіменування вказівника	
	&	Отримати адресу	
	(тип)	Перетворення типу	
	sizeof	Визначення розміру операнда в байтах	
3	*	Множення	Зліва направо
	/	Ділення	
	%	Остача від ділення (застосовується до операндів цілого типу даних)	
4	+	Додавання	Зліва направо
	-	Віднімання	
5	<<	Побітовий зсув вліво	Зліва направо
	>>	Побітовий зсув вправо	
6	<	Операція відношення «Менше»	Зліва направо
	<=	Операція відношення «Менше або дорівнює»	
	>	Операція відношення «Більше»	
	>=	Операція відношення «Більше або дорівнює»	
7	==	Операція відношення «Дорівнює»	Зліва направо
	!=	Операція відношення «Не дорівнює»	
8	&	Побітове «I» (побітова кон'юнкція)	Зліва направо

Продовження табл. 3.10

Ранг операції	Позначення операції	Опис операції	Асоціативність
9	\wedge	Побітове «Виключне АБО» (додавання за модулем 2)	Зліва направо
10	$ $	Побітове «АБО» (побітова диз'юнкція)	Зліва направо
11	$\& \&$	Операція «Логічне І»	Зліва направо
12	$\ $	Операція «Логічне АБО»	Зліва направо
13	$? :$	Тернарна умовна операція	Справа наліво
14	$=$	Присвоєння	Справа наліво
	$+ =$	Додавання з присвоєнням	
	$- =$	Віднімання з присвоєнням	
	$* =$	Множення з присвоєнням	
	$/ =$	Ділення з присвоєнням	
	$\% =$	Обрахунок остачі від ділення з присвоєнням	
	$\& =$	Побітове «І» з присвоєнням	
	$\wedge =$	Побітове «Виключне АБО» з присвоєнням	
	$ =$	Побітове «АБО» з присвоєнням	
	$<<=$	Побітовий зсув вліво з присвоєнням	
	$>>=$	Побітовий зсув вправо з присвоєнням	
15	,	Кома	Зліва направо

В табл. 3.10 окремі відмінні між собою операції позначаються однаковим символом, наприклад, « $\&$ » або « $*$ », або « $-$ », — відповідними символами можуть позначатися унарні операції та бінарні операції. Видно, що, наприклад, унарна операція $\&$ (отримати адресу) має вищий пріоритет, ніж бінарна операція $\&$ (побітове «І»), або унарна операція « $-$ » (унарний «мінус») має вищий пріоритет, ніж бінарна операція « $-$ » (віднімання). Таким чином, при записі відповідного виразу або оператора необхідно правильно обирати

потрібну операцію. В табл. 3.10 круглі дужки () представлені в якості операції виклику функцій, а також операції перетворення типу, крім того, круглі дужки можуть використовуватися в виразах для формування послідовності виконання дій аналогічно як в математиці. Поряд з іншими, в табл. 3.10 представлені логічні операції, а також операції відношення, результатом обчислення яких є логічний (булевий) тип даних, при якому результат може набувати одного із двох можливих значень — логічна «істина» (true — англійською мовою) та логічна «хиба» (false — англійською мовою). В мові програмування С логічну «хибу» позначають числовим значенням 0. А логічна «істина» позначається значенням 1 (або будь-яким ненульовим значенням). Наприклад, логічний вираз, сформований із операції відношення «Більше» $5 > 10$ — хибний, тому резултатом цього виразу буде 0 (логічна «хиба»), а вираз, в якому використана операція «Менше»: $5 < 10$ — істинний, тому резултатом цього виразу буде 1 (логічна «істина»). Нижче подано приклад використання відповідних операцій, що представлені в табл. 3.10 [5—7].

Ранг 1.

- 2.1 () операція виклику функцій. Застосовується разом з іменем функції. Наприклад, ця операція використовувалася разом з вище розглянутими функціями printf() чи scanf() при їх виклику в програмі.
- 2.2 [] операція індексації масиву, використовується при роботі із одновимірними чи багатовимірними масивами, в квадратних дужках записується індекс елементу масиву при виконанні певних дій над масивами.
- 2.3 . операція використовується при роботі із структурним типом даних для вибору елементу структури.
- 2.4 -> операція використовується при роботі із структурним типом даних для непрямого вибору елементу структури.

Ранг 2.

1.1 ! операція логічне «НЕ» (логічне заперечення)

$!1 = 0$ (в даному випадку 1 розглядається як логічна «істина». Логічне заперечення «істини» в результаті дає логічну «хибу»)

$!2 = 0$ (значення 2 розглядається як логічна «істина». Логічне заперечення «істини» в результаті дає логічну «хибу»)

$!(-5) = 0$ (значення -5 розглядається як логічна «істина». Логічне заперечення «істини» в результаті дає логічну «хибу»)

$!0 = 1$ (значення 0 розглядається як логічна «хиба». Логічне заперечення «хшиби» в результаті дає логічну «істину»)

В контексті написання програмного коду:

```
int a = 0;  
int b;  
  
b = !a;  
  
printf("b=%d", b); // на екрані буде b=1
```

Змінна **b** буде набувати значення, що дорівнює 1.

```
int c = 1;  
int d;  
  
d = !c;  
printf("d=%d", d); // на екрані буде d=0
```

Змінна **d** буде набувати значення, що дорівнює 0.

1.2 \sim операція побітової інверсії — біти із значенням 0 замінюються на значення 1, а біти із значенням 1 замінюються на 0.

```
char a = -128;  
char b;  
  
b = ~a;  
printf("b=%d", b); // на екрані буде b = 127
```

Значення змінної `a = -128` — це від'ємне ціле число, що зберігається в доповняльному коді, двійковий код для числа -128 такий:

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

Після інверсії розрядів двійкового коду значення бітів буде таким:

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

Отриманому двійковому коду відповідає десяткове число $+127$.

Тому, за результатами виконання операції `~` що була застосована до змінної `a` з подальшим збереженням отриманого результату в змінній `b`, змінна `b` буде зберігати значення $+127$.

1.3 `++` інкремент — збільшення значення змінної на 1.

Операція «інкремент» має дві форми — префіксну і суфіксну.

В *префіксній* формі символ операції інкременту записується перед іменем змінної. Особливість операції інкременту в префіксній формі полягає в тому, що спочатку виконується оновлення значення змінної, а потім змінна із оновленим значенням використовується у виразі:

```
int x = 5;
int y;

y = ++x;

printf("y=%d    x=%d", y, x); // виводиться y=6  x=6
```

В *суфіксній* формі символ операції інкременту записується після імені змінної. Особливість операції інкременту в суфіксній формі полягає в тому, що оновлення змінної відбудеться після опрацювання виразу:

```
int x = 5;
int y;

y = x++;

printf("y=%d    x=%d", y, x); // виводиться y=5  x=6
```

1.4 -- декремент — зменшення значення змінної на 1.

Операція «декремент» має дві форми — префіксну і суфіксну, які працюють аналогічним чином, що і описана вище операція «інкремент».

1.5 + унарний «плюс».

```
int x = +5;
```

В явному вигляді вказується, що змінній `x` присвоюється значення `+5`.

1.6 - унарний «мінус».

```
int x = 20;
int y;
```

```
y = -x;
```

```
printf("y=%d x=%d", y, x); //виводиться y=-20 x=20
```

Зміна знаку на протилежний. Змінна `y` отримує значення `-20`.

1.7 * операція розіменування (або іншими словами — доступ за адресою, що зберігається в операнді, до якого застосовується дана операція). Використовується при роботі із вказівниками.

1.8 & операція отримання адреси (наприклад, застосовується при роботі з функцією `scanf()`, при зчитуванні значення змінної, тим самим дана операція дозволяє отримати адресу змінної в пам'яті).

1.9 (*тип*) операція перетворення (приведення) типів. Може застосовуватися, наприклад, в арифметичних виразах, для перетворення типів даних операндів, що приймають участь в обчисленнях, з метою коректного виконання розрахунків.

```
float y, x;
int a=5, b=2;
```

```
y = a/b;
```

```

x = (float)a / (float)b;

printf("y=% .2f  x=% .2f", y, x); // y=2.00  x=2.50

```

1.10 `sizeof` операція визначення кількості байтів, що займає в пам'яті операнд. Може застосовуватися до *типу даних* або, наприклад, до змінної. Якщо операція застосовується до *типу даних*, то тип даних треба записувати в дужках.

```

unsigned int x;

x = sizeof (char);

printf("x=%u", x); // виводиться x=1

```

Змінна `x` отримує значення кількості байтів, яка виділяється в системі для типу даних `char`.

```

unsigned int y;
double a;

y = sizeof a;

printf("y=%u", y); // виводиться y=8

```

Змінна `y` отримує значення кількості байтів, яка виділяється для типу даних `double`, при цьому, на відміну від попереднього варіанту, операція застосовується до змінної `a` типу `double`.

Ранг 3.

2.1 * операція множення.

```

float x = 2.0, y = 5.5;
float result = x * y; // змінній result присвоюється
                      // результат множення x * y

printf("result = %.1f", result); // виводиться result = 11.0

```

2.2 / операція ділення. Якщо в операції ділення операнди мають цілий тип даних, результат ділення не буде містити дробової частини (дробова частина відсікається). В разі необхідності виконувати ділення над цілими типами даних, щоб не втратити дробову частину,

необхідно виконувати явне перетворення типів даних, застосовуючи операцію, яка була розглянута в пункті 2.9. Нижче приводяться варіанти застосування операції ділення, коли в якості операндів виступають змінні цілого типу і не виконується перетворення типів даних (*варіант а*); і також варіант, коли в якості операндів виступають змінні цілого типу і застосовується операція явного перетворення типів даних (*варіант б*). Крім того, приводиться варіант, коли обидва операнди в операції ділення мають дійсний тип даних (тип даних **float**) (*варіант в*). В кожному із варіантів, результат зберігається в змінній дійсного типу (тип даних **float**). Потрібно пам'ятати, якщо дійсне значення, яке має цілу та дробову частини, присвоювати змінній цілого типу, то в цьому випадку, змінна цілого типу буде зберігати лише цілу частину, а дробова частина буде відсікатися (а не округлюватися).

а) відсікається дробова частина

```
int a = 3, b = 10;
float c;

c = b / a; // При діленні відсікається дробова частина.
            // 10/3 = 3 Ціле значення 3 перетворюється до
            // дійсного значення 3.0 і присвоюється змінній c

printf("c = %.3f", c ); // виводиться c = 3.000
```

б) використовується операція явного перетворення (приведення) типів для коректного виконання операції ділення і збереження дробової частини в отриманому результаті.

```
int a = 3, b = 10;
float c;

c = (float)b/ (float)a;
                    // відбувається перетворення типів в
                    // операції ділення. Цілі значення 10 та
                    // 3 приводяться до дійсного типу float
                    // і в операції ділення застосовуються
                    // дійсні значення 10.0 та 3.0. Результат
                    // буде дійсне число 3.333333, яке
                    // присвоюється змінній c, яка має тип
                    // даних float
printf("c = %.3f", c ); // виводиться c = 3.333
```

в) ділення відбувається над дійсними числами

```
float a = 3.0, b = 10.0;  
float c;  
  
c = b / a;  
  
printf("c = %.3f", c); // виводиться c = 3.333
```

2.3 % операція визначення остачі від ділення. Дано операція потребує використання операндів цілих типів.

```
int a = 4, b = 14;  
int c;  
  
c = b % a;  
  
// Принцип визначення результату операції «Остача від  
// ділення» можна представити наступним чином:  $\frac{14}{4} = 3 \frac{2}{4}$ .  
// В результаті, остача буде дорівнювати 2  
  
printf("c = %d", c); // виводиться c = 2
```

Ранг 4.

3.1 + операція додавання. Виконується звичайним чином як в математиці.

3.2 - операція віднімання. Виконується звичайним чином як в математиці.

Ранг 5.

4.1 << операція побітового зсуву вліво.

```
int a = 3, b = 12;  
int c;  
  
c = b << a;  
  
printf("c = %d", c); // виводиться c = 96
```

В даному прикладі відбувається побітовий зсув двійкового коду значення, що збережне в зміній b (змінна b зберігає значення 12) на 3 розряди вліво (дана кількість зберігається в змінній a).

Отриманий в ході таких дій двійковий код, буде визначати відповідне десяткове число, що буде зберігатися в змінній с.

Отже, двійковий код значення 12:

0	0	0	0	1	1	0	0
---	---	---	---	---	---	---	---

Після зсуву на 3 розряди вліво:



Таким чином, двійковий код буде такий:

0	1	1	0	0	0	0	0
---	---	---	---	---	---	---	---

При перетворенні отриманого двійкового коду в десяткову систему числення для заданого прикладу отримаємо остаточне значення, що буде дорівнювати +96.

4.2 >> операція побітового зсуву вправо. Данна операція працює схожим чином як показано в попередньому пункті. Тільки в даному випадку розряди зсуваються вправо на задану кількість позицій.

Ранг 6.

5.1 < операція «менше». Операція відношення «менше» часто застосовується в логічних виразах, в виразі-умові, де, наприклад, необхідно порівняти між собою два значення. Результатом операції є логічна «хиба» або логічна «істина» (логічний «хибі» відповідає числове значення 0, логічний «істині» відповідає ненульове значення, наприклад, 1).

```
int x;  
x = 15 < 9; // Результат операції логічна «хиба»  
// змінна x отримає значення 0.
```

```
printf("x = %d", x); // виводиться x = 0
```

Інший варіант:

```
int x;  
x = 2 < 99; // Результат операції логічна «істина»  
// змінна x отримає значення 1.  
printf("x = %d", x); // виводиться x = 1
```

- 5.2 \leq операція «менше або дорівнює». Операція працює аналогічним чином, що і в попередньому пункті, за виключенням того, що якщо обидва операнди будуть рівними, то результат операції буде логічна «істина».
- 5.3 $>$ операція «більше». Якщо перший операнд більший за другий — результат операції логічна «істина», в іншому випадку — логічна «хиба».
- 5.4 \geq операція «більше або дорівнює». Якщо значення першого операнда більше за значення другого операнда, або якщо значення першого і другого операндів рівні між собою — результат операції логічна «істина», в іншому випадку — логічна «хиба».

Ранг 7.

- 6.1 $==$ операція «дорівнює». Якщо обидва операнди мають однакові значення, то результат операції — логічна «істина», якщо операнди мають різні значення, то результат операції — логічна «хиба». *Важливо* не плутати операцію «дорівнює» (яка позначається $==$) із операцією присвоєння (яка позначається $=$).

```
int x;  
x = 2 == 99; // Результат операції логічна «хиба»  
// змінна x отримає значення 0.  
printf("x = %d", x); // виводиться x = 0
```

або інший варіант:

```
int x;  
x = 34 == 34; // Результат операції логічна «істина»  
// змінна x отримає значення 1.  
printf("x = %d", x); // виводиться x = 1
```

6.2 `!=` операція «Не дорівнює». Якщо обидва операнди мають різні значення, то результат операції — логічна «істина», якщо значення операндів одинакові, то результат операції — логічна «хиба».

```
int x;
x = 34 != 34; // Результат операції логічна «хиба»
                // змінна x отримає значення 0.
printf("x = %d", x); // виводиться x = 0
```

Ранг 8.

7.1 `&` операція побітове «І» (побітова кон'юнкція). Якщо значення відповідних бітів обох операндів дорівнює 1, то значення відповідного результуючого біту буде також 1. В усіх інших випадках значення відповідного результуючого біту в двійковому коді отриманого результату буде дорівнювати 0.

```
char x = 29;
char y = -104;
char t;

t = x & y;

printf("t=%d", t); // виводиться t = 24
```

В наведеному прикладі, двійковий код для значення змінної `x` (яка дорівнює 29) буде таким (прямий код):

0	0	0	1	1	1	0	1
---	---	---	---	---	---	---	---

Двійковий код значення для значення змінної `y` (яка дорівнює -104) буде таким (доповняльний код):

1	0	0	1	1	0	0	0
---	---	---	---	---	---	---	---

Двійковий код результата буде визначатися шляхом взаємодії відповідних бітів двійкових кодів змінних x та y :

&	0	0	0	1	1	1	0	1
	1	0	0	1	1	0	0	0
Результат:	0	0	0	1	1	0	0	0

Знайдений двійковий код в ході застосування операції побітового «І» буде таким: 00011000_2 . Це прямий код. При переведенні двійкового коду в десяткову систему числення, отримано значення 24_{10} .

Таким чином, змінна t в наведеному вище фрагменті коду буде отримувати значення 24, що і буде виводитися в консольне вікно.

Ранг 9.

8.1 ^ операція побітове «Виключне АБО». Якщо відповідні біти обох операндів мають однакові значення, то значення відповідного результируючого біту буде дорівнювати 0. В усіх інших випадках значення відповідного результируючого біту буде дорівнювати 1.

```
char x = 29;
char y = -104;
char t;

t = x ^ y;

printf("t=%d", t); // виводиться t = -123
```

В наведеному прикладі, двійковий код для значення змінної x (яка дорівнює 29) буде таким (прямий код):

0	0	0	1	1	1	0	1
---	---	---	---	---	---	---	---

Двійковий код значення для значення змінної y (яка дорівнює -104) буде таким (доповняльний код):

1	0	0	1	1	0	0	0
---	---	---	---	---	---	---	---

Двійковий код результата буде визначатися шляхом взаємодії відповідних бітів двійкових кодів змінних x та y :

	0	0	0	1	1	1	0	1
1	0	0	1	1	0	0	0	0
Результат:	1	0	0	0	0	1	0	1

Знайдений двійковий код в ході застосування операції «Виключне АБО» буде таким: 10000101_2 . Оскільки в операції приймали участь змінні знакового цілого типу даних `char`, то отриманий двійковий код буде інтерпретуватися як доповняльний код (оскільки в найстаршому (знаковому) розряді стоїть 1, це значить що значення від'ємне, а від'ємні значення зберігаються в доповняльному коді). При переведенні отриманого двійкового коду в десяткову систему числення, буде отримано значення -123_{10} .

Таким чином, змінна t в наведеному вище фрагменті програмного, коду буде отримувати значення -123_{10} .

Ранг 10.

9.1 | операція побітове «АБО» (побітова диз'юнкція). Якщо відповідні біти обох операндів мають значення 0, то значення відповідного результиручого біту буде дорівнювати 0. В усіх інших випадках значення відповідного результиручого біту буде дорівнювати 1.

```
char x = 29;
char y = -104;
char t;

t = x | y;

printf("t=%d", t); // виводиться t = -99
```

В наведеному прикладі, двійковий код для значення змінної x (яка дорівнює 29) буде таким (прямий код):

0	0	0	1	1	1	0	1
---	---	---	---	---	---	---	---

Двійковий код значення для значення змінної y (яка дорівнює -104) буде таким (доповняльний код):

1	0	0	1	1	0	0	0
---	---	---	---	---	---	---	---

Двійковий код результата буде визначатися шляхом взаємодії відповідних бітів двійкових кодів змінних x та y :

	0	0	0	1	1	1	0	1
	1	0	0	1	1	0	0	0
Результат:	1	0	0	1	1	1	0	1

Знайдений двійковий код в ході застосування операції побітове «АБО» буде таким: 10011101_2 . Оскільки в операції приймали участь змінні знакового цілого типу даних `char`, то отриманий двійковий код буде інтерпретуватися як доповняльний код (оскільки в найстаршому (знаковому) розряді стоїть 1, це значить що значення від'ємне, а від'ємні значення зберігаються в доповняльному коді). При переведенні отриманого двійкового коду в десяткову систему числення, отримано значення -99_{10} .

Таким чином, в наведеному вище фрагменті програми змінна t буде отримувати значення -99_{10} , що і буде виводитися в консольне вікно.

Ранг 11.

10.1 `&&` операція логічне «І». Ця операція використовується, наприклад, для формування логічних виразів (виразів-умови) в умовних операторах. Якщо обидва операнди мають значення логічна «істина», то результат операції буде логічна «істина». В усіх інших випадках — результат буде логічна «хиба». Оскільки для вираження логічної «істини» та логічної «хиби» використовуються числові значення, то логічній «хибі» відповідає значення 0, а логічній «істинні» відповідає ненульове значення (значення 1).

```

int a = 5, b = 10;
int r;

r = (a<b) && (a!=24);

printf("r=%d", r); // виводиться r = 1 (логічна «істина»)

```

Вираз `(a<b)` буде мати результат логічна «істина» (або 1 в числовому вираженні), оскільки 5 менше 10. Вираз `(a!=24)` буде мати результат логічна «істина» (або 1 в числовому вираженні), оскільки 5 не дорівнює 24. Таким чином, для операції логічне «`&&`» операнд зліва від операції має значення логічна «істина»; операнд, що стоїть справа від операції, також має значення логічна «істина», тому остаточний результат в ході виконання операції `&&` буде логічна «істина».

Якби хоч один із операндів в операції логічне «`&&`» мав би значення логічна «хиба», то і остаточний результат виконання операції логічне «`&&`» дорівнював би логічній «хибі».

Ранг 12.

11.1 `||` операція логічне «АБО». Ця операція також як і попередня операція часто використовується для формування логічних виразів (виразів-умови) в умовних операторах. Якщо обидва операнди даної операції мають значення логічна «хиба», то результат операції буде логічна «хиба». В усіх інших випадках — результат буде логічна «істина».

```

int a = 5, b = 10;
int r;

r = (a>100) || (b==a);

printf("r=%d", r); // виводиться r = 0 (логічна «хиба»)

```

Вираз `(a>100)` буде мати результат логічна «хиба» (значення 0 в числовому вираженні), оскільки «5 більше ніж 100» — це хибне

твірдження. Вираз `(b==a)` буде мати результат логічна «хиба» (значення 0 в числовому вираженні), оскільки «10 дорівнює 5» — це хибне твірдження. Таким чином, для операції логічне «АБО» операнд зліва від операції має значення логічна «хиба», операнд, що стоїть справа від операції, також має значення логічна «хиба», тому остаточний результат в ході виконання операції `||` буде логічна «хиба».

Якби хоча б один із операндів в операції логічне «АБО» мав би значення логічна «істина», то і остаточний результат виконання операції логічне «АБО» дорівнював би логічній «істині».

Ранг 13.

12.1 `? :` тернарна умовна операція. Ця операція має наступний формат запису:

Вираз №1 `?` *Вираз №2* `:` *Вираз №3*;

Якщо *Вираз №1* має результат логічна «істина», то результатом операції `? :` буде значення, яке обраховується у *Виразі №2*.

Якщо *Вираз №1* має результат логічна «хиба», то результатом операції `? :` буде значення, яке обраховується у *Виразі №3*.

```
int a = 5, b = 10;  
int r;  
  
r = (a>b) ? a : b;  
  
printf("r=%d", r); // виводиться r = 10
```

В наведеному прикладі змінній `r` присвоюється більше із двох значень, що зберігаються у змінніх `a` та `b`.

Якщо вираз `(a>b)` — істинний, то змінній `r` присвоюється значення змінної `a`.

Якщо вираз $(a>b)$ — хибний, то змінній r присвоюється значення змінної b .

Для заданого прикладу, вираз $(a>b)$ в результаті обчислення набуває значення логічна «хиба», оскільки « $5>10$ » — це хибне твердження. Тому змінній r присвоюється результат обчислення *Виразу №3* (відповідно до формату запису операції $?:$), а *Виразу №3* складається лише із змінної b . Тому змінній r буде присвоюватися значення змінної b .

Ранг 14.

13.1 = операція присвоєння. Використовується для присвоєння певного значення, наприклад, змінній або елементу масиву тощо.

```
int a = 5, b = 10;
```

В наведеному прикладі змінній a присвоюється значення 5, змінній b присвоюється значення 10. Важливо не плутати операцію присвоєння та операцію відношення «дорівнює».

В табл. 3.10, в стовпці «Асоціативність» для операції присвоєння вказано «Справа наліво», тобто якщо у виразі зустрічається декілька операцій присвоєння, то вони виконуються справа наліво, наприклад, це дозволяє записувати такі вирази по наданню відповідних значень деяким змінним:

```
int a, b, c;  
a=b=c=25;
```

Спочатку змінній c присвоюється значення 25. Результатом присвоєння є значення, яке присвоюється, таким чином змінні b таож присвоюється значення 25. Змінній a також присвоюється значення 25.

13.2 += операція додавання з присвоєнням. Відноситься до виду складених операцій присвоєння. Наприклад:

```
int b = 10;
```

```

b += 5; // b = b + 5

printf("b=%d", b); // виводиться b = 15

```

Загалом, всі складені операції мови С можна схематично представити в такому виді:

$$a \circ= b$$

де \circ позначає відповідний знак операції. Цей вираз можна переписати у наступному вигляді (що демонструє особливість запису складених операцій присвоєння):

$$a = a \circ b$$

Таким чином, складену операцію додавання (операція «додавання з присвоєнням»):

$$a += b$$

можна переписати так:

$$a = a + b$$

Аналогічним способом виконуються і всі інші складені операції присвоєння 14-го рангу.

Ранг 15.

14.1 , операція «кома». Використовується для виконання послідовності виразів. Вирази, які записуються через кому обчислюються відповідно до заданого правила асоціативності, тобто зліва направо. Значення та *тип* даних результата виразу, що розміщений на крайній правій позиції, буде розглядатися як остаточний результат.

```

int a, b, x, y;
a = 3;
x = 8;
y = -1;
b = (a*x+4, 10-x, 10%3+1, a*x*y);

printf("b=%d", b); // виводиться b = -24

```

В представленому прикладі спочатку визначається результат обчислень, що записані в круглих дужках, а потім отримане значення присвоюється змінній **b**. Розглянемо окремо запис в круглих в дужках:

$$(a * x + 4, 10 - x, 10 \% 3 + 1, a * x * y)$$

Спочатку визначається результат обрахунку $a * x + 4$ (отримане значення буде дорівнювати 28). Потім обраховується вираз $10 - x$ (отримане значення буде дорівнювати 2). Потім обраховується вираз $10 \% 3 + 1$ (отримане значення буде дорівнювати 2). Після цього обраховується вираз $a * x * y$ (отримане значення буде дорівнювати -24), і саме цей останній результат буде остаточним результатом в ході виконання дій, які записані в круглих дужках. Таким чином, змінній **b** буде присвоюватися значення -24.

3.4 Питання для самоконтролю

1. Навіщо потрібно підключати в програму бібліотечні заголовочні файли?
2. Яким чином в програмі можна створити константу за допомогою директиви препроцесора `#define`?
3. Яка відмінність між ініціалізацією змінної і оголошенням змінної?
4. Яка різниця між типами даних `int` та `unsigned int`?
5. Що таке ранг операції?
6. Яка різниця між операціями, які позначаються символом `=` та `==`?
7. Яка відмінність між операціями інкременту (декременту), що записуються в суфіксній та префіксній формі?
8. Яке числове значення має логічна «хиба»?

4. УМОВНІ ОПЕРАТОРИ

При написанні програм, які реалізують відповідні алгоритми, дуже часто виникає необхідність виконувати певний набір дій тільки, якщо виконані певні умови, наприклад, в алгоритмі вирішення квадратного рівняння знаходити квадратний корінь тільки, якщо значення дискримінанту невід'ємне число, або, наприклад, в деякій арифметичній операції виконувати операцію ділення, тільки у випадку, якщо дільник не дорівнює 0, або, наприклад, присвоювати деякій змінній, яка зберігає значення екзаменаційного балу студента, значення балу при умові, якщо цей бал перевищує певну мінімальну допустиму величину. Таким чином, постійно доводиться зіштовхуватися із відповідними умовами, виконувати відповідні перевірки, виконувати опрацювання відповідних логічних виразів, і в залежності від їх результату виконувати той або інший набір операторів.

До умовних операторів (тобто, до операторів які передбачають аналіз певної умови, що представлена у вигляді певного логічного виразу) відносять оператори вибору, — такі оператори ще називають операторами розгалуження або операторами вибору. Це такі оператори [3—9]:

- 1) `if`
- 2) `if/else`
- 3) `switch`

Далі буде розглянуто особливість кожного із таких операторів та надані приклади їх застосування.

4.1 Умовний оператор вибору `if`

Умовний оператор вибору `if` передбачає спочатку визначення значення відповідного виразу-умови, який записується в круглих дужках біля службового слова `if`. Вираз-умова — розглядається як логічний вираз,

результат якого може дорівнювати одному із двох можливих значень — логічній «істині» або логічній «хибі». Якщо результат виразу-умови — логічна «істина», тоді виконуються відповідні оператори які відносяться до оператора if. Якщо результат виразу-умови — логічна «хиба», тоді оператори, які відносяться до if, ігноруються, а продовжується виконання операторів, що йдуть далі по тексту програми після умовного оператора if. Принцип роботи умовного оператора if можна проілюструвати за допомогою блок-схеми, яка показана на рис. 4.1 [3, 4].

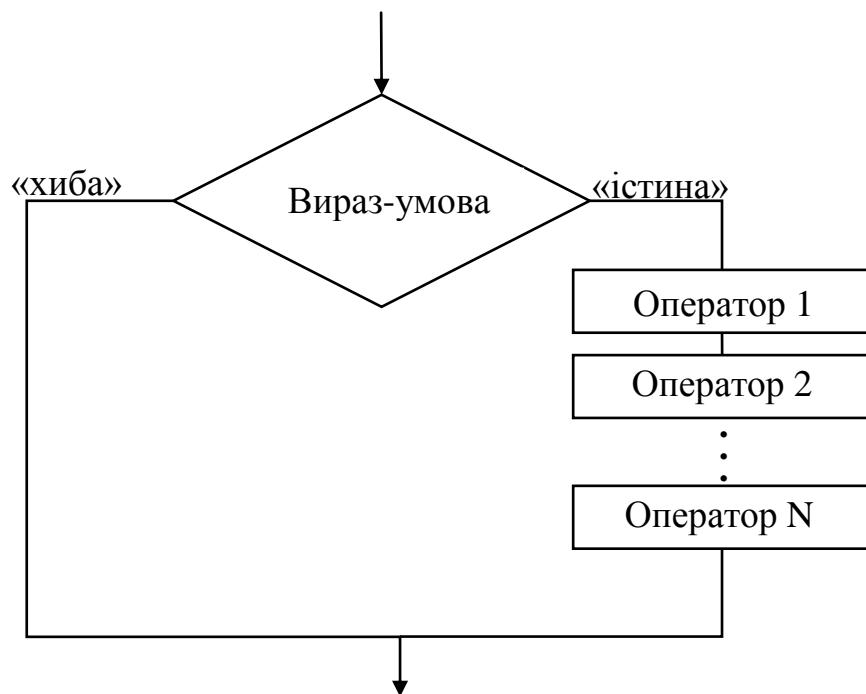


Рис. 4.1 — Блок-схема умовного оператора if

Блок-схемі, що зображена на рис.4.1, буде відповідати наступний псевдо-код, який демонструє певний шаблон за яким може виконуватися написання умовного оператора if в програмі [5—7]:

```

if ( вираз-умова )
{
    Оператор1;
    Оператор2;
    ...
    ОператорN;
}
  
```

В наведеному прикладі, набір операторів Оператор₁,...,Оператор_N виконується тільки у випадку істинності виразу-умови. Якщо результатом виразу-умова буде логічна «хиба», тоді ніяких дій не виконується, і відбувається перехід до виконання наступних операторів, які записані після умовного оператора if.

Потрібно пам'ятати, що в мові програмування С, логічна «хиба» має відповідний числовий еквівалент — це значення 0. Логічна «істина» в якості відповідного числового еквіваленту може розглядати будь-яке ненульове значення. Таким чином, в якості виразу-умови можуть бути записані, наприклад, арифметичні вирази, і у випадку, якщо результатом арифметичного виразу буде 0 — тоді такий вираз-умова буде сприйматися хибним, а якщо результатом буде ненульове значення, наприклад, 1, 5, -4, — тоді результатом такого виразу-умови буде логічна «істина». Такий підхід дозволяє в якості виразу-умови в операторі if записувати, наприклад, лише ім'я певної змінної, і в залежності від значення, що зберігається в такій змінній, — умова буде інтерпретуватися «хибною» (якщо значення змінної дорівнює 0) або «істинною» (якщо значення змінної дорівнює ненульовій величині). В якості виразу умови можуть використовуватися не тільки вирази, що утворені із застосуванням операцій відношення: «більше», «більше або дорівнює», «менше», «менше або дорівнює», «дорівнює», «не дорівнює», але можуть створюватися більш різноманітні та складні вирази-умови.

В наведеній на рис. 4.1 блок-схемі оператора if показано, що у випадку істинності виразу-умови, необхідно виконувати набір операторів. В наведеному вище псевдо-коді для оператора if видно, що цей набір операторів, який виконується в разі істинності виразу-умови, повинен бути записаний між двома фігурними дужками. Ці дві фігурні дужки — відкриваюча фігурна дужка «{« та закриваюча фігурна дужка «»}» формують блок. Потрібно запам'ятати, що у випадку, коли необхідно виконувати більше ніж один оператор, при істинності виразу-умови, потрібно відповідні оператори записувати у вигляді блоку.

В мові програмування С оператор — це певна інструкція або команда, яка повинна бути виконана. Більшість операторів при їх записі потребують в кінці ставити символ «крапка з комою».

Якщо передбачається, що при істинності виразу-умови, буде виконуватися лише один оператор, тоді блок-схема оператора if буде мати вигляд як на рис. 4.2 [3, 4].

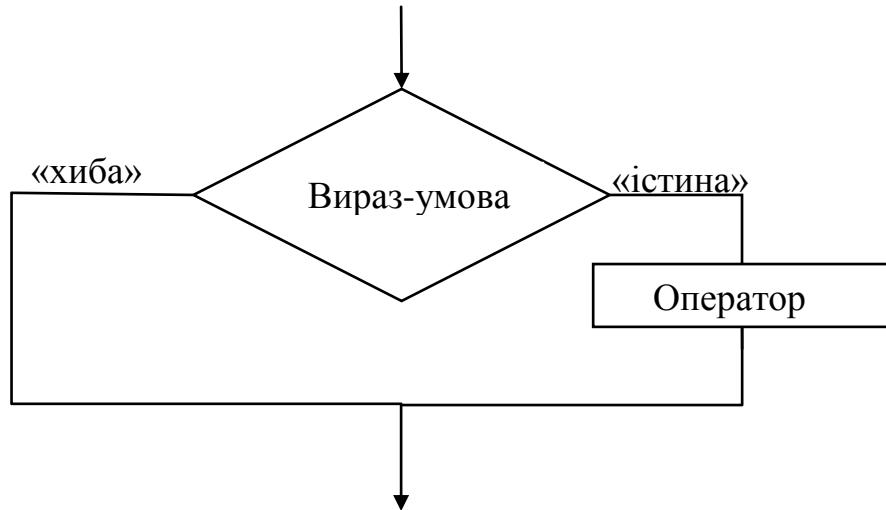


Рис. 4.2 — Блок-схема умовного оператора if

І в такому випадку, фігурні дужки можна не записувати [5—8]:

```
if ( вираз-умова )
    Оператор;
```

Але можна залишати фігурні дужки, формуючи блок, навіть, якщо передбачається виконувати тільки один оператор у випадку істинності виразу-умови:

```
if ( вираз-умова )
{
    Оператор;
}
```

Такі окремі деталі щодо оформлення коду можуть бути узгоджені заздалегідь, щоб забезпечувати одинаковий стиль написання коду програми. Можна помітити, що оформлення коду програми може також забезпечуватися використанням пробільних символів (символу «пробіл» або «табуляція»), щоб

робити текст програми більш зручним для візуального сприйняття та аналізу коду.

Важливо пам'ятати, що після виразу-умови, що записується у круглих дужках біля службового слова `if`, символ «крапка з комою» ставити не потрібно.

Крім відступів, можуть також застосуватися різні стилі проставлення фігурних дужок, крім наведеного вище варіанту, може також застосовуватися такий варіант:

```
if ( вираз-умова ) {  
    Оператор1;  
    Оператор2;  
    ...  
    ОператорN;  
}
```

Якщо для випадку, коли при істинності виразу-умови, повинні виконуватися декілька операторів (які мають бути записані між відкриваючою та закриваючою фігурними дужками), забути записати ці фігурні дужки як в прикладі нижче:

```
if ( вираз-умова )  
    Оператор1;  
    Оператор2;  
    ...  
    ОператорN;
```

то в такому випадку оператори, починаючи з Оператор`2` до Оператор`N` вже відношення до оператора `if` не мають. І не зважаючи на те, що може здатися нібито за рахунок зроблених відступів в тексті програми, при істинності виразу-умови будуть виконуватися всі оператори — починаючи з Оператор`1` і закінчуєчи Оператор`N`, але насправді до оператора `if` буде мати відношення тільки Оператор`1`, а інші — до оператора `if` відношення не мають ніякого, і будуть виконуватися незалежно від значення виразу-умови. Ілюстрація такого поводження при виконанні коду програми, може бути представлена наступним варіантом запису псевдо-коду, в якому за допомогою блоку, сформовано із

фігурних дужок, показного який саме оператор буде відноситися до if (це буде лише Оператор1):

```
if ( вираз-умова ) {  
    Оператор1;  
}  
  
Оператор2;  
...  
ОператорN;
```

Дуже часто пропуск відкриваючої та закриваючої фігурних дужок при формуванні блоку операторів, які мають виконуватися в разі істинності виразу-умови в if, є логічною помилкою. Крім того, варто пам'ятати що після виразу-умови, який записується в круглих дужках біля if, крапку з комою ставити не потрібно. Якщо поставити після виразу-умови символ «крапка з комою», то, враховуючи, що «крапка з комою» позначає пустий оператор, при істинності виразу-умови буде виконуватися саме пустий оператор, а той фрагмент коду, який по початковій задумці розробника мав би виконуватися при істинності виразу-умови в if, до оператора if вже не буде мати ніякого відношення. Приклад такого показано нижче:

```
if ( вираз-умова );  
{  
    Оператор1;  
    Оператор2;  
    ...  
    ОператорN;  
}
```

Цей фрагмент псевдо-коду можна переписати в наступному вигляді, акцентувавши увагу на тому, що саме пустий оператор (який позначений символом «крапка з комою») буде мати відношення до оператора if, в той час як блок операторів починаючи з Оператор1 і закінчуєчи ОператорN, вже до if не відноситься, і ці оператори будуть виконуватися після того, які виконається умовний оператор if:

```

if( вираз-умова )
;
    // виконується пустий оператор
    // якщо результат виразу-умови буде
    // логічна «істина»

{
    // цей блок до if не відноситься
Оператор1;
Оператор2;
...
ОператорN;
}

```

Приклад використання умовного оператора `if` наведено у представленому нижче програмному коді. Оператор `if` використовується для аналізу значення змінної `grade`, яка призначена для зберігання екзаменаційного балу студента. Значення змінної `grade` вводиться з клавіатури. Якщо введене значення виходить за діапазон 60...100, то програма виводить в консольне вікно повідомлення «Недопустимі вхідні дані», і відбувається завершення програми за допомогою виклику функції `exit()`. Для використання функції `exit()` необхідно підключити за допомогою директиви препроцесора `#include` заголовочний файл стандартної бібліотеки `stdlib.h`. Функція `exit()` потребує передачу їй параметру. Для випадку, коли передбачається, що завершення програми шляхом виклику функції `exit()` відбувається в штатному режимі, то функції `exit()` передається параметр 0 (або може записуватися константа `EXIT_SUCCESS`). Для випадку, коли передбачається, що завершення програми шляхом виклику функції `exit()` відбувається при неможливості далі продовжити виконання програми і потрібно аварійно її завершити, то функції `exit()` передається параметр 1 (або в якості параметра функції `exit()` може записуватися константа `EXIT_FAILURE`).

Для того, щоб мати можливість виводити в консольне вікно текст, що написаний буквами українського алфавіту, необхідно підключити заголовочний файл `locale.h`, а в програмі викликати функцію `setlocale(LC_CTYPE, "Ukrainian")`. Однак при цьому, при наборі тексту для

його подальшого відображення у консольному вікні, літеру українського алфавіту «і» потрібно замінювати на літеру англійського алфавіту «і» (при роботі із консольним вікном операційної системи Windows).

Якщо значення для змінної `grade` було введене коректно, програма аналізує значення змінної, у випадку, якщо значення змінної знаходиться в діапазоні 60...64 в консольне вікно буде виводитися текст: Достатньо. Це буде оцінка, що відповідає введенному балу. Якщо значення змінної `grade` буде в діапазоні 65...74, повідомлення, що виводиться в консольне вікно буде: Задовільно. Якщо `grade` буде в діапазоні 75...84, програма буде виводити на екран монітору текст: Добре. У випадку, якщо `grade` потрапляє в проміжок 85...94, в консольному вікні буде виведено: Дуже Добре. І якщо значення змінної `grade` буде більше або рівне 95, то буде виводитися текст: Відмінно. Текстові позначення оцінок: Достатньо, Задовільно, Добре, Дуже Добре, Відмінно збережені як константні текстові рядки із використанням директиви препроцесора `#define`. Для цього використані відповідні константи, що мають імена: `SUFFICIENT`, `SATISFACTORY`, `GOOD`, `VERYGOOD`, `EXCELLENT`. Для більш зручного візуального сприйняття і аналізу тексту програми, при оформлені коду були зроблені відповідні відступи за допомогою пробільних символів. Використання пробільних символів не впливає на виконання коду програми, а лише дозволяє більш якісно виконувати оформлення тексту програми.

Дана програма ілюструє використання умовного оператора `if` для контролю вхідних даних, а також використання операцій відношення та операцій логічне «І» та логічне «АБО» для формування виразу-умови у відповідних операторах `if` на етапі аналізу значень змінної `grade`.

В наведеному нижче коді програми, що реалізує поставлене завдання, у випадку істинності умови в першому умовному операторів `if`, який зустрічається в тексті програми, виконуються два оператори: `printf("\nНедопустимі вхідні дані!!!\n");` та `exit(EXIT_SUCCESS);` — для

формування блоку операторів використовуються фігурні дужки. В тих умовних операціях if, при істинності виразу-умови яких, передбачається виконання лише одного оператора, — виклику функції printf(), — фігурні дужки не записувалися, оскільки вони в цьому випадку є необов'язковими:

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

#define SUFFICIENT      "Достатньо"
#define SATISFACTORY   "Задовільно"
#define GOOD            "Добре"
#define VERYGOOD        "Дуже Добре"
#define EXCELLENT       "Відмінно"

int main()
{
    int grade;

    setlocale(LC_CTYPE, "Ukrainian");
    printf("\nВведіть бал. Допустимий діапазон 60...100: ");
    scanf("%d", &grade);

    if( grade < 60 || grade > 100 )
    {
        printf("\nНедопустимі вхідні дані!!!\n");
        exit(EXIT_SUCCESS);
    }

    printf("\nОцінка: ");

    if( grade >= 95 )
        printf("%s", EXCELLENT);

    if( grade >= 85 && grade < 95 )
        printf("%s", VERYGOOD);

    if( grade >= 75 && grade < 85 )
        printf("%s", GOOD);

    if( grade >= 65 && grade < 75 )
        printf("%s", SATISFACTORY);

    if( grade >= 60 && grade <= 64 )
        printf("%s", SUFFICIENT);

    printf("\n\n");
    return 0;
}
```

На рис. 4.3 показано вигляд консольного вікна при введенні з клавіатури значення 95.

```
Введіть бал. Допустимий діапазон 60...100: 95
Оцінка: Відмінно

Process returned 0 <0x0> execution time : 3.459 s
Press any key to continue.
```

Рис. 4.3 — Виведення результату виконання програми в консольне вікно

Аналізуючи код програми, можна помітити, що, наприклад, для заданого значення 95, після виконання обчислення виразу-умови в операторі if: `if(grade >= 95)` і виведення на екран текстового повідомлення Відмінно, перевірка виразу-умови відбувається в наступних операторах if, тобто в тих, в яких для заданого значення 95, умови будуть хибними. Враховуючи, що всі умови, які записані в операторах if для аналізу значення змінної `grade`, є взаємовиключні, то доцільно було б, що як тільки обчислення виразу-умови дає логічну «істину» всі інші вирази-умови в тих операторах if, що залишились, — не перевіряти. Наприклад, для заданого значення 95 було б доцільно не проводити подальше обчислення логічних виразів для кожного оператора if, що записані нижче по тексту після `if(grade >= 95)`. Якби, наприклад, було задано значення 87, то доцільно було б виконати перевірку виразу-умови в операторі `if(grade >= 95)` і визначивши, що результат є логічною «хибою», перейти до наступної перевірки виразу-умови в наступному операторі if: `if(grade >= 85 && grade < 95)`, і визначивши, що в цьому операторі if вираз-

умова дає логічну «істину», не переходити до наступних операторів if, оскільки в них відповідні вирази-умови будуть хибними.

Такий підхід можна реалізувати шляхом використання умовного оператора `if/else`.

4.2 Умовний оператор вибору `if/else`

На відміну від оператора if, у якого відповідно до блок-схеми, показаної на рис. 4.1, набір операторів виконується в разі істинності виразу-умови, в операторі if/else міститься альтернативний набір операторів, що виконуються в разі хибності логічного виразу, який записується в круглих дужках біля службового слова if. Приклад блок-схеми умовного оператора if/else показано на рис. 4.4 [3, 4].

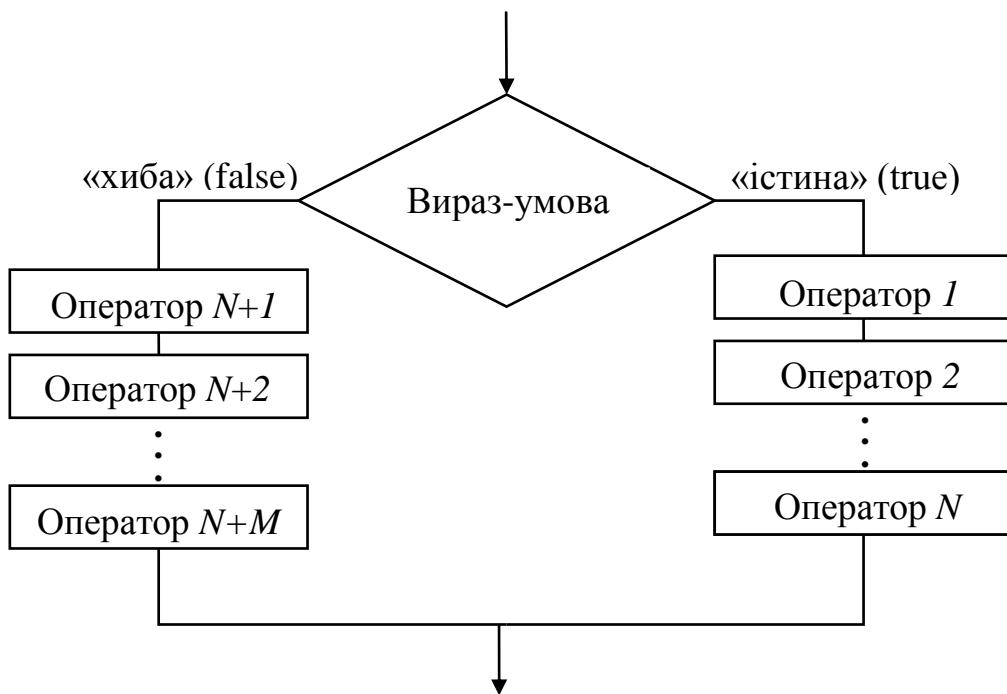


Рис. 4.4 — Блок-схема умовного оператора `if/else`

У вигляді псевдо-коду оператор `if/else` може бути записаний так:

```

if ( вираз-умова )
{
    // виконуються, якщо результат вираза-умови — «істина»
    Оператор1;
    Оператор2;
    ...
    ОператорN;
}
else
{
    // виконуються, якщо результат вираза-умови — «хиба»
    ОператорN+1;
    ОператорN+2;
    ...
    ОператорN+M;
}

```

Аналогічно оператору if, в операторі if/else фігурні дужки можна не використовувати, якщо планується виконання лише одного оператора в залежності від результату виразу-умови, наприклад [5—8]:

```

if ( вираз-умова )
    Оператор1;
else
    Оператор2;

```

Потрібно звертати увагу, щоб після круглих дужок, в яких записаний вираз-умова, не ставився символ «крапка з комою». В іншому випадку — це може привести до помилки на етапі компіляції проекту. Якщо, відповідно до рис. 4.4, в одній із гілок блок-схеми оператора if/else — в основній гілці (коли вираз-умова дорівнює «істині») або в альтернативній (коли вираз-умова дорівнює «хибі»), — повинен виконуватися лише один оператор, тоді фігурні дужки при записі такого єдиного оператора, можна опустити, наприклад:

```

if ( вираз-умова )
{
    Оператор1;
    Оператор2;
    ...
    ОператорN;
}
else
    ОператорN+1;

```

Або

```
if ( вираз-умова )
    Оператор1;
else
{
    Оператор2;
    Оператор3;
    ...
    ОператорN;
}
```

В наведених прикладах запису оператора if/else фігурні дужки спеціально були записані на окремих рядках, щоб акцентувати на них увагу. При написані коду програми може застосовуватися інший стиль форматування тексту в частині розташування фігурних дужок:

```
if ( вираз-умова ) {
    Оператор1;
    Оператор2;
    ...
    ОператорN;
} else {
    ОператорN+1;
    ОператорN+2;
    ...
    ОператорN+M;
}
```

або такий варіант:

```
if ( вираз-умова ) {
    Оператор1;
    Оператор2;
    ...
    ОператорN;
}
else {
    ОператорN+1;
    ОператорN+2;
    ...
    ОператорN+M;
}
```

Потрібно брати до уваги, що умовні оператори if та if/else можуть містити вкладені оператори if та/або if/else, і при відсутності форматування тексту відповідно до прийнятих правил, при відсутності проставлення відступів і відсутності застосування єдиного стилю проставлення фігурних дужок там, де вони потрібні, — все це може суттєво ускладнити написання, налагодження та внесення змін в програмний код. Тому важливо притримуватися відповідного стилю при написанні власних програм. В іншому випадку, аналіз програмного коду сторонніми експертами, наприклад, з метою оптимізації алгоритму, пошуку помилок в коді програми, удосконалення раніше реалізованих рішень, — може перетворитися на вкрай складний процес, що буде потребувати значних затрат часу.

Приклад блок-схеми, на якій представлено частину деякого алгоритму, що містить вкладені оператори if/else, показано на рис. 4.5. Покладається, що всі змінні, які використовуються були раніше оголошені, а деякі з них попередньо ініціалізовані. Наведена блок-схема носить лише демонстраційний характер застосування вкладених умовних операторів.

Загалом, на етапі аналізу алгоритму, перевірки правильності його роботи, а також на етапі його тестування, побудова блок-схеми алгоритму може суттєво спрощувати ці дії, оскільки дає змогу візуалізувати шляхи подальшого руху по програмному коду, який повинен виконуватися при істинності (чи хибності) тієї або іншої умови, яка використовується в умовних операторах. Враховуючи, що реальні алгоритми можуть мати досить велику кількість вкладених умовних операторів, то часто без блок-схеми розібрatisя в деталях реалізації алгоритму може бути досить обтяжливим. Тому, незважаючи на те, що для простих задач, побудова блок-схеми може видаватися збитковим та непотрібним етапом про розробці та аналізі алгоритму, але, разом з тим, це може суттєво поліпшити процес виправлення повилок та внесення змін в алгоритм на етапі його модифікації.

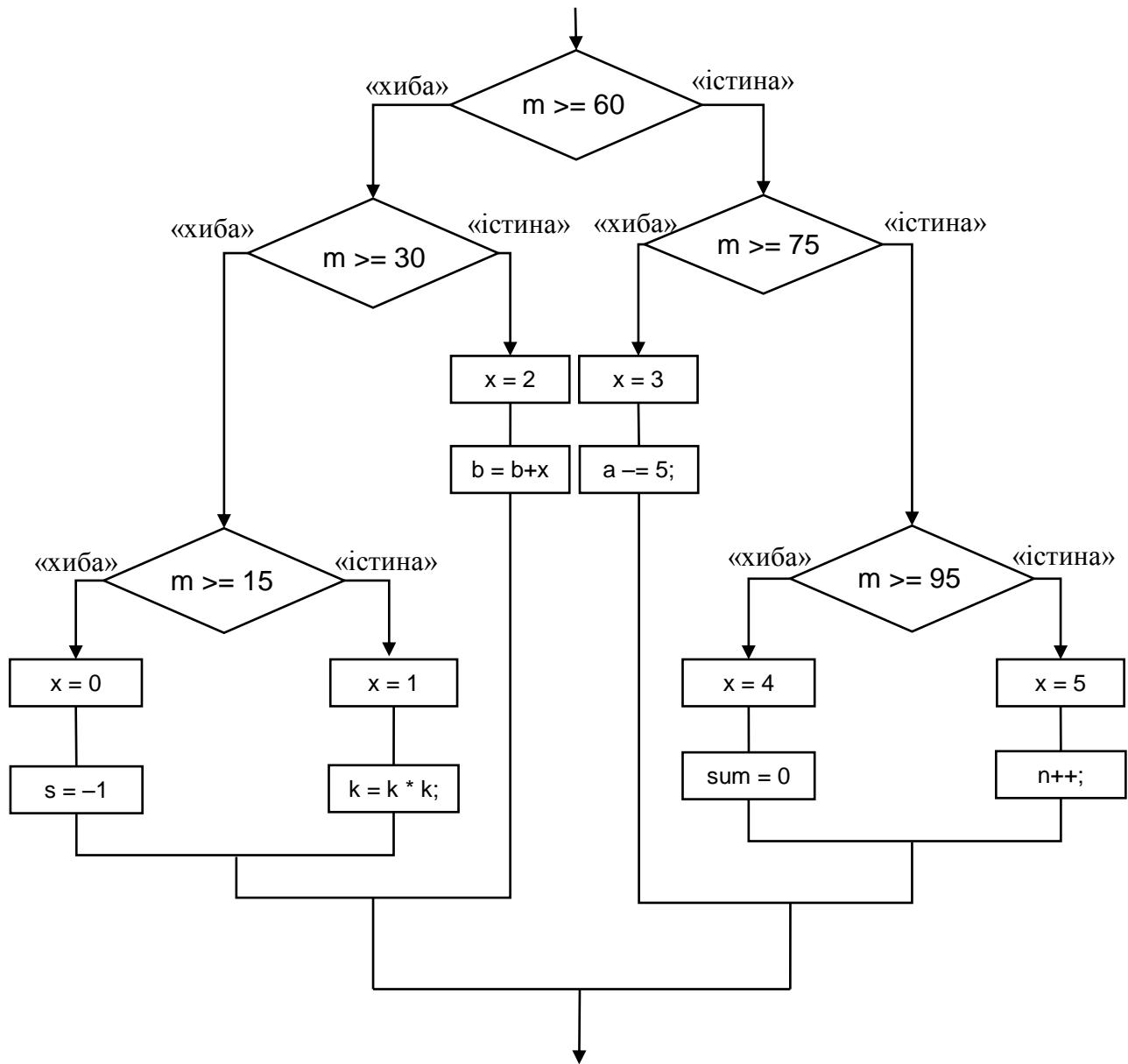


Рис. 4.5 — Фрагмент блок-схеми алгоритму, що містить вкладені умовні оператори if/else

Програмний код, який відповідає блок-схемі, що приведена на рис. 4.5, представлений нижче, і може слугувати лише одним із можливих варіантів форматування тексту програми з метою поліпшення його читабельності. Існують різні стилі оформлення тексту програми. Головне, обравши певний стиль, притримуватися його при написанні коду для всього проекту. Гарно оформленний і відформований текст програми дозволяє суттєво полегшити

виявлення помилок в коді програми або подальшу модифікацію коду програми. Наведений приклад оформлення коду носить лише ілюстративний характер:

```
if ( m >= 60 )
    if ( m >= 75 )
        if ( m >= 95 ) {
            x = 5;
            n++;
        }
        else {
            x = 4;
            sum = 0;
        }
    else {
        x = 3;
        a -= 5;
    }
else
    if ( m >= 30 ) {
        x = 2;
        b = b + x;
    }
else
    if ( m >= 15 ) {
        x = 1;
        k = k * k;
    }
    else {
        x = 0;
        s = -1;
    }
```

В наведеному прикладі можна побачити, що службові слова `if` та `else`, які формують умовний оператор `if/else` знаходяться на одному рівні. Таким чином, візуально набагато легше зрозуміти який фрагмент коду виконується при істинності або при хибності того логічного виразу, який записується в круглих дужках біля відповідного `if`. Окрім того, враховуючи що `if/else` вважається одним оператором, тому фігурні дужки для формування блоку не застосовувалися в тих місцях коду, де в разі істинності або хибності відповідного логічного виразу, планується виконання лише одного вкладеного оператора `if/else`. Це дозволяє не перевантажувати текст програми фігурними дужками в тих місцях тексту програми, де в них немає необхідності. З іншого, боку, якщо при написанні коду програми групою розробників прийнято використовувати всюди в умовних операторах фігурні дужки для формування

блоку операторів, тоді наведений фрагмент коду міг би виглядати наступним чином:

```
if ( m >= 60 ) {
    if ( m >= 75 ) {
        if ( m >= 95 ) {
            x = 5;
            n++;
        } else {
            x = 4;
            sum = 0;
        }
    } else {
        x = 3;
        a -= 5;
    }
} else {
    if ( m >= 30 ) {
        x = 2;
        b = b + x;
    } else {
        if ( m >= 15 ) {
            x = 1;
            k = k * k;
        } else {
            x = 0;
            s = -1;
        }
    }
}
```

В другому варіанті можна помітити, що відповідна закриваюча фігурна дужка знаходиться на одному рівні із найближчим службовим словом if оператора if/else до якого відповідна фігурна дужка відноситься, тобто формує блок операторів, який виконується в залежності від істинності чи хибності відповідних логічних виразів, які наведені в прикладі. Такий підхід також дозволяє досить просто ідентифікувати принадлежність фрагменту програми до того чи іншого умовного оператора, якщо при реалізації алгоритму виникає необхідність використовувати велику кількість вкладених умовних операторів.

Може бути використаний трохи змінений варіант оформлення тексту програми при написанні коду, який відноситься до **else**-частини оператора if/else:

```

        if ( m >= 60 ) {
            if ( m >= 75 ) {
                if ( m >= 95 ) {
                    x = 5;
                    n++;
                }
            }
            else {
                x = 4;
                sum = 0;
            }
        }
        else {
            x = 3;
            a -= 5;
        }
    }
    else {
        if ( m >= 30 ) {
            x = 2;
            b = b + x;
        }
        else {
            if ( m >= 15 ) {
                x = 1;
                k = k * k;
            }
            else {
                x = 0;
                s = -1;
            }
        }
    }
}

```

В представленому варіанті оформлення тексту програми службове слово **else** стоїть на окремому рядку під відповідною закриваючою фігурною дужкою, яка закриває блок оператора **if/else**, який виконується у випадку істинності виразу-умови, що розташований у круглих дужках біля службового слова **if**, а закриваюча фігурна дужка, яка формує блок, що відноситься до **else**-частини, розміщується на окремому рядку під відповідним службовим словом **else**.

При написанні коду програми можуть використовуватися інші, ніж продемонстровані вище, варіанти стилю оформлення коду програми. Головне — при написанні програми всюди притримуватися того стилю форматування тексту програми, який обраний за домовленістю групою розробників, або

такого стилю, який вважається загальновживаним, оскільки якісно оформленій код програми дозволяє поліпшити ефективність налагодження програми.

З урахуванням розглянутих особливостей оператора if/else, програма по аналізу вхідного балу студента, яка була продемонстрована вище, може бути переписана таким чином:

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

#define SUFFICIENT      "Достатньо"
#define SATISFACTORY    "Задовільно"
#define GOOD            "Добре"
#define VERYGOOD         "Дуже Добре"
#define EXCELLENT        "Відмінно"

int main()
{
    int grade;
    setlocale(LC_CTYPE, "Ukrainian" );
    printf("\nВведіть бал. Допустимий діапазон 60...100: ");
    scanf("%d", &grade );

    if( grade < 60 || grade > 100 )
    {
        printf("\nНедопустимі вхідні дані!!!\n");
        exit(EXIT_SUCCESS);
    }
    else
    {
        printf("\nОцінка: ");

        if( grade >= 95 )
            printf("%s", EXCELLENT );

        else if( grade >= 85 && grade < 95 )
            printf("%s", VERYGOOD );

        else if( grade >= 75 && grade < 85 )
            printf("%s", GOOD );

        else if( grade >= 65 && grade < 75 )
            printf("%s", SATISFACTORY );

        else
            printf("%s", SUFFICIENT );
        printf("\n\n");
    }
    return 0;
}
```

Блок-схема алгоритму програми може виглядати так як показано на рис. 4.6 (змінна **grade** на елементах блок-схеми відображеня як символ g).

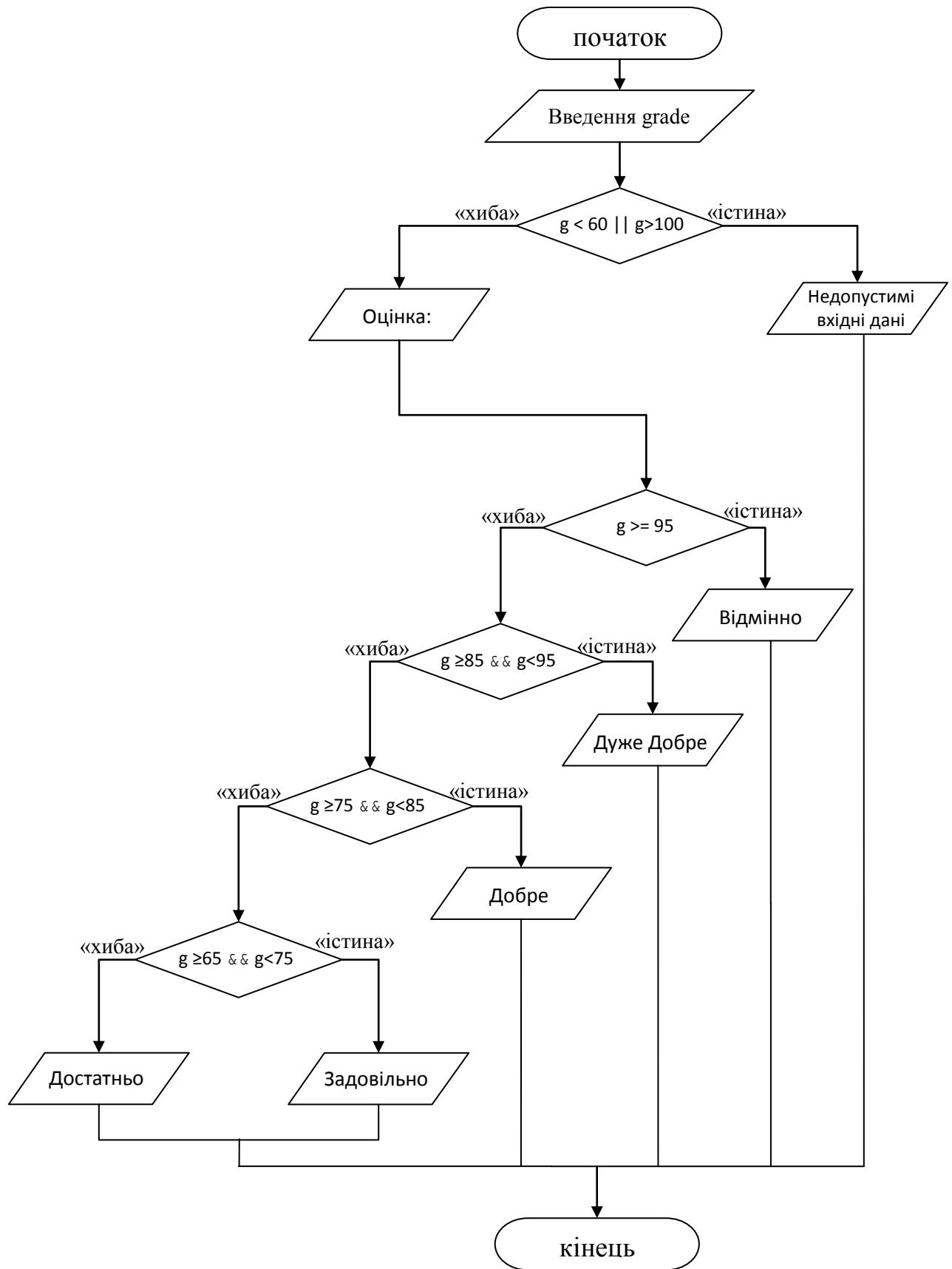


Рис. 4.6 — Блок-схема алгоритму

4.3 Оператор **switch**

Оператор switch має таку структуру запису [5, 7]:

```
switch ( вираз )
{
    case Константа1:
        оператор1;
        оператор2;
        ...
        операторN;
        break;

    case Константа2:
        операторN+1;
        операторN+2;
        ...
        операторN+K;
        break;

    и т.д.
    .
    .
    .

    case КонстантаL:
        операторS;
        операторS+1;
        ...
        операторM;
        break;

    default:
        операторM+1;
        операторM+2;
        ...
        операторM+T;
}
```

Оператор switch функціонує наступним чином: спершу виконується визначення значення виразу, який записується в круглих дужках біля службового слова **switch**. В якості виразу може записуватися арифметичний вираз або, наприклад,

ім'я змінної, яка зберігає певне значення. Єдина вимога, щоб значення виразу було значенням цілого типу.

Оператор switch в своєму записі містить службове слово `case` поряд з яким через пробіл записується значення константи, яка повинна мати цілий тип. В наведеній вище схемі запису такі константи були позначені як Константа₁, Константа₂, ..., Константа_L. Ці константи називаються мітками. Значення виразу, що записане в круглих дужках біля службового слова `switch`, порівнюється із значенням Константа₁ — якщо ці значення співпадають, то виконуються оператори які записані після `case` Константа₁, причому виконання операторів відбувається до того моменту поки не зустрінеться оператор `break`, який дозволить вийти із оператора `switch`, після чого продовжується виконання операторів, які йдуть в програмі після оператора `switch`. Оператор `break` може бути відсутнім, — в такому разі виконання операторів буде продовжуватися, поки не буде досягнуто закриваючої фігурної дужки, після цього в програмі продовжиться виконання операторів, які записані після `switch`. Якщо значення виразу в круглих дужках біля службового слова `switch` не дорівнює значенню Константа₁, тоді відбувається перевірка чи співпадає значення виразу із значенням Константа₂, — якщо значення співпадають, то виконання починається з операторів, які записані після `case` Константа₂ і до того моменту поки не зустрінеться оператор `break`, який дозволить вийти із оператора `switch`, або якщо оператор `break` відсутній, то виконання операторів буде продовжуватися поки не буде досягнуто завершення оператора `switch`. Якщо значення виразу, що записано в круглих дужках біля службового слова `switch`, не дорівнює значенню Константа₂, то відбувається перевірка чи співпадає значення виразу із значенням Константа₃ і процес виконується аналогічно описаній вище процедурі.

Якщо виявиться, що значення виразу не співпадає із жодною міткою (жодною Константою), тоді виконуються оператори, які записані після службового слова `default`.

Варіант `default` є необов'язковим, тому він може бути відсутнім в операторі `switch`. Якщо, наприклад, варіант `default` — відсутній, а значення виразу не співпадає із жодною міткою (жодною Константою), тоді оператор `switch` разом із записаними в ньому операторами — пропускається.

Якщо потрібно виконувати один і той самий набір дій для випадку, коли значення виразу в круглих дужках біля `switch`, буде співпадати з різними значеннями міток, тоді замість того, щоб дублювати відповідний фрагмент коду дляожної такої мітки, можна відповідним чином послідовно розмістити мітки, наприклад, як схематично показано нижче:

```
switch( вираз )
{
    case Константа1:
    case Константа2:
    case Константа3:
    case Константа4:
        оператор1;
        оператор2;
        ...
        операторN;
        break;

    case Константа5:
    case Константа6:
    case Константа7:
    case Константа8:
        операторN+1;
        операторN+2;
        ...
        операторN+K;
        break;

    default:
        операторM+1;
        операторM+2;
        ...
        операторM+T;
}
```

В наведеній схемі, якщо значення виразу буде дорівнювати значенню Константа1 або Константа2 або Константа3 або Константа4, то в цьому випадку

буде виконуватися набір операторів оператор 1 , оператор $2 \dots$ оператор N . Якщо значення виразу буде дорівнювати значенню Константа 5 або Константа 6 або Константа 7 або Константа 8 , то в цьому випадку буде виконуватися набір операторів оператор $N+1$, оператор $N+2 \dots$ оператор $N+M$. А якщо значення виразу не співпаде із жодною із вказаних констант, то тоді будуть виконуватися оператори, що відносяться до default — це оператори оператор $M+1$, оператор $M+2 \dots$ оператор $M+T$.

А якщо, наприклад, оператори **break** відсутні [5]:

```
switch( вираз )
{
    case Константа1:
    case Константа2:
    case Константа3:
    case Константа4:
        оператор1;
        оператор2;
        ...
        операторN;

    case Константа5:
    case Константа6:
    case Константа7:
    case Константа8:
        операторN+1;
        операторN+2;
        ...
        операторN+K;

    default:
        операторM+1;
        операторM+2;
        ...
        операторM+T;
}
```

і якщо значення виразу співпало із значенням, наприклад, Константа 1 , то будуть виконуватися всі оператори — починаючи від оператор 1 до оператор $M+T$. Нижче продемонстровано приклад застосування оператора switch. В якості виразу, значення якого порівнюється із мітками, — виступає значення змінної var. Якщо змінна var зберігає значення 1, тоді виконується

обчислення квадратного кореня із x (для цього використовується функція стандартної бібліотеки `sqrt()`). Щоб мати можливість викликати цю функцію потрібно підключити заголовочний файл `math.h`). Передбачено, що перед викликом функції `sqrt()` виконується перевірка значення змінної x , і якщо значення змінної від'ємне, то програма повідомляє про недопустимість обраної операції і завершується. Якщо змінна дорівнює 2, то визначається — парне чи непарне значення змінної x . Якщо змінна `var` дорівнює 3, то значення змінної x підноситься до квадрату. Для всіх інших значень змінної `var` в консольне вікно виводиться інформація про неправильний вибір варіанту дій, і програма на цьому завершується. Для реалізації сформульованого завдання використовуються вкладені оператори `switch`. В одному варіанті в якості виразу для аналізу у вкладеному операторів `switch` використовується логічний вираз `x >= 0`, який може мати одне з двох можливих значень — логічна «істина» (таким чином, результат виразу дорівнює 1) або логічна «хиба» (таким чином, результат виразу дорівнює 0). У виразі `x%2` для іншого оператора `switch` визначається остача від ділення значення змінної x на 2 — якщо змінна x зберігає парне число, то результат виразу `x%2` буде дорівнювати 0, а якщо змінна x зберігає непарне число, то результат виразу `x%2` буде дорівнювати 1. В залежності від значень відповідних виразів виконуються варіанти `case 0` або `case 1` у відповідних вкладених операторах `switch`.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <math.h>

int main()
{
    int x;
    int var;
    double result;
    setlocale( LC_CTYPE, "Ukrainian" );

    printf("Введіть значення параметра x");
    printf("\n");
    scanf("%d", &x);

    printf("\nОберіть варіант дій:\n");
    printf("1 - sqrt(x)\n");
    printf("2 - x^2\n");
    printf("3 - sqrt(x)^2\n");
    printf("0 - exit\n");
}
```

```

printf("2 - перевірити x на парність/непарність\n");
printf("3 - піднести x до квадрату");
printf("\n>");
scanf("%d", &var);

switch( var )
{
    case 1:
        switch( x >= 0 )
        {
            case 0:
                printf("\nЗаборонена операція !!!");
                printf("\nx повинен бути більшим або рівним 0");
                printf("\nЗавершення програми.\n");
                exit(EXIT_SUCCESS);
                break;

            case 1:
                result = sqrt(x);
                printf("\nsqrt(x)=% .3lf", result);
                break;
        }
        break;

    case 2:
        switch( x%2 )
        {
            case 0:
                printf("\nx=%d - це ПАРНЕ число\n", x);
                break;

            case 1:
                printf("\nx=%d - це НЕПАРНЕ число\n", x);
                break;
        }
        break;

    case 3:
        x *= x;
        printf("\nРезультат: %d \n", x);
        break;

    default:
        printf("\nНеправильний вибір !!!");
        printf("\nЗавершення програми.\n");
        exit(EXIT_SUCCESS);
}

return 0;
}

```

На рис. 4.7 показано результат виконання програми при заданих значеннях $x = 25$, $var = 1$.

```
Введіть значення параметра x
>25

Оберіть варіант дій:
1 - sqrt(x)
2 - перевірити x на парність/непарність
3 - піднести x до квадрату
>1

sqrt(x)=5,000
Process returned 0 <0x0>  execution time : 2.691 s
Press any key to continue.
```

Рис. 4.7 — Результат виконання програми при $x = 25$, $var = 1$

На рис. 4.8 показано результат виконання програми при заданих значеннях $x = -9$, $var = 1$.

```
Введіть значення параметра x
>-9

Оберіть варіант дій:
1 - sqrt(x)
2 - перевірити x на парність/непарність
3 - піднести x до квадрату
>1

Заборонена операція !!!
x повинен бути більшим або рівним 0
Завершення програми.

Process returned 0 <0x0>  execution time : 3.022 s
Press any key to continue.
```

Рис. 4.8 — Результат виконання програми при $x = -9$, $var = 1$

4.4 Оператор циклу `for`

Якщо потрібно щоб в програмі деякий набір операторів виконувався певну задану кількість разів або поки певний логічний вираз, який виступає в якості виразу-умови, є істинним, то в цьому випадку потрібно використовувати оператори циклу, зокрема до операторів циклу відноситься оператор `for`. В найбільш загальному вигляді, оператор `for` може бути записаний так:

```
for ( ініціалізація; вираз-умова; оновлення )  
{  
    оператори, які виконуються в циклі;  
}
```

Оператори, які виконуються в циклі — формують тіло циклу.

Вираз *ініціалізація* зазвичай передбачає надання початкового значення певній змінній, яка використовується надалі у *виразі-умові*, а також така змінна може використовуватися у виразі *оновленні*.

Якщо результатом *виразу-умови* є логічна «істина», то відбувається прохід по циклу, тобто відбувається виконання операторів, які формують тіло циклу. Один прохід по циклу, тобто одне виконання тіла циклу, називається *ітерацією*. Після виконання однієї ітерації, відбувається виконання виразу *оновлення*, потім відбувається визначення результата *виразу-умови*, і якщо результат обрахунку *виразу-умову* є логічна «істина», то відбувається виконання наступної ітерації циклу, тобто відбувається виконання операторів, які формують тіло циклу, а якщо результатом обрахунку *виразу-умови* буде логічна «хиба», то відбувається вихід із циклу і виконання переходить до операторів, які знаходяться після циклу. Якщо при вході в цикл результат *виразу-умови* сразу є хибним, то цикл не виконується, і програма переходить до виконання операторів, які знаходяться після циклу. При записі оператора циклу, можливий такий варіант запису фігурних дужок, які формують блок [4]:

```
for ( ініціалізація; вираз-умова; оновлення ) {  
    оператори, які виконуються в циклі;  
}
```

Якщо в тілі циклу передбачається виконання лише одного оператора, тоді фігурні дужки можна не використовувати [4, 6]:

```
for ( ініціалізація; вираз-умова; оновлення )  
    деякий_один_оператор;
```

Розглянемо застосування оператора циклу `for` для обчислення суми елементів числового ряду, який має наступний вид:

$$1, 3, 5, 7, \dots (2N-1)$$

потрібно знайти суму:

$$\text{Sum} = 1 + 3 + 5 + 7 + \dots + (2N-1),$$

де `Sum` — змінна, яка зберігає обраховану суму, `N` — параметр, який визначає кількість елементів числового ряду. Для наведеного прикладу вважається, що значення елементів числового ряду є без знаковими цілими числами.

В наведеному нижче прикладі реалізації сформульованої задачі по знаходженню суми представленого числового ряду, вважається, що параметр `N` задається користувачем з клавіатури. При цьому, враховуючи, що кількість елементів ряду — це величина невід'ємна, тому при написанні програми необхідно передбачити виконання відповідних перевірок на етапі введення параметру `N`, і у випадку, коли користувач вводить від'ємне значення передбачити відповідні дії, наприклад, сповістити користувача про неприпустимість введеного значення і завершити програму. Для параметру `N` обрано тип даних `int`. Враховуючи, що сума елементів числового ряду в наведеному прикладі, буде невід'ємно, тому для змінної `Sum`, яка буде використовуватися в процесі знаходження результату, а також буде зберігати знайдену суму, доцільно використати тип даних `unsigned int`. В програмі викликається функція `getch()`, яка використовується для очікування натискання користувачем будь-якої клавіші при затримці текстової інформації в консольному вікні. Фактично, функція `getch()` призначена для зчитування символу при натисканні відповідної клавіші клавіатури при роботі в консольному вікні. Функція `getch()` повертає значення коду символу в кодовій

таблиці ASCII. Однак в наведеному прикладі, зчитаний цією функцією символ (точніше — код символу із таблиці ASCII), не присвоюється ніякій змінній, тобто функція використовується лише для призупинення виконання наступного фрагменту коду програми, очікуючи натискання від користувача клавіші на клавіатурі. Для того, що скористатися функцією `getch()` необхідно в програму підключити заголовочний файл `conio.h`.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <conio.h>
int main()
{
    unsigned int x;      // поточне значення елементу числового ряду
    unsigned int Sum;   // сума значень елементів числового ряду
    int N;              // кількість елементів числового ряду

    setlocale(LC_CTYPE, "Ukrainian"); //для відображення символів кирилиці
    printf("Для продовження необхідно задати значення для змінної N.\n");
    printf("\nЗначення N повинно бути цілим числом більше НУЛЯ.");
    printf("\nЯкщо задати N <= 0 — програма завершиться.\n");
    printf("\nВведіть N: ");

    printf("\nN=");
    scanf("%d", &N);

    if( N <= 0 ){
        printf("\n\nНатисніть будь-яку клавішу для завершення програми... ");
        getch();
        exit(0);
    }
    else {
        printf("\nОбраховується сума значень:\n");

        Sum = 0;
        for ( x = 1; x <= (2*N-1); x += 2 ) {
            Sum += x;

            // відображення на екрані значень ряду, які підсумовуються
            if( x < (2*N-1) )
                printf("%u + ", x);
            else
                printf("%u", x);
        }
    }

    printf("\n\n ----- Результат обрахунку -----");
    printf("\nСума = %u\n", Sum);

    printf("\n\nНатисніть будь-яку клавішу для завершення програми... \n");
    getch();

    return 0;
}
```

Замість використання функції `getch()` для забезпечення паузи у виконанні програми до натискання користувачем будь-якої клавіші, можна також скористатися командою `system("pause")`, при цьому необхідно підключити заголовочний файл `stdlib.h`.

На рис. 4.9 показано результат виконання програми при $N = 5$.

```
Для продовження необхідно задати значення для змінної N.  
Значення N повинно бути цілим числом більше НУЛЯ.  
Якщо задати N <= 0 - програма завершиться.  
Введіть N:  
N=5  
Обраховується сума значень:  
1 + 3 + 5 + 7 + 9  
----- Результат обрахунку -----  
Сума = 25  
  
Натисніть будь-яку клавішу для завершення програми...
```

Рис. 4.9 — Результат виконання програми при $N = 5$

Оператор циклу `for` дозволяє досить різноманітно підходити до свого запису. Наприклад, вираз *ініціалізація* може бути записаний перед заголовком циклу, а вираз *новлення* може записуватися в тілі циклу, схематично це можна показати таким чином [5]:

```
ініціалізація;  
for ( ; вираз-умова; )  
{  
    оператори, які виконуються в циклі;  
    оновлення;  
}
```

Навіть вираз-умова може не записуватися взагалі [5]:

```
ініціалізація;  
for ( ; ; )  
{  
    оператори, які виконуються в циклі;  
    оновлення;  
}
```

В такому випадку (якщо поле для *виразу-умови* — пусте), то вважається, що значення *виразу-умови* — логічна «істина», і в цьому випадку цикл буде

виконуватися нескінченну кількість разів (якщо не передбачено інші способи завершення його виконання).

4.5 Оператор циклу `while`

Крім оператора циклу `for`, в мові програмування С доступний також ще один оператор циклу — `while`.

В загальному випадку, цикл `while`, записується в коді програми таким чином [3, 6]:

```
while ( вираз-умова )
{
    оператори, які виконуються в циклі;
}
```

Аналогічно розглянутому вище оператору `for`, формат запису фігурних дужок, за допомогою яких формується блок (тіло циклу), може бути таким:

```
while ( вираз-умова ) {
    оператори, які виконуються в циклі;
}
```

Якщо в циклі повинен виконуватися лише один оператор, то фігурні дужки які формують блок, можна опустити. В такому випадку, оператор циклу `while` буде записуватися в коді програми відповідно до шаблону:

```
while ( вираз-умова )
    деякий оператор, який виконується в циклі;
```

Загалом, оператора циклу `while` може бути цілком достатньо, щоб виконувати реалізацію відповідних завдань, де необхідно певний набір команд виконувати ітеративно (тобто повторювати певний набір дій деяку кількість разів). Наприклад, фрагмент коду програми, що включає в себе оператор `for`, який в загальному випадку записаний у вигляді:

```

for ( Ініціалізація; Вираз-умова; Оновлення )
{
    Оператори, які виконуються в циклі;
}

```

може бути переписаний шляхом використання оператора циклу while. В такому випадку записаний вище в загальному виді фрагмент коду, при заміні оператора **for** на оператор **while**, буде мати наступний вигляд:

```

Ініціалізація;
while ( Вираз-умова )
{
    Оператори, які виконуються в циклі;
    Оновлення;
}

```

Блок-схема оператора **while**, яка демонструє принцип його функціонування, показана рис. 4.10 [4, 9].

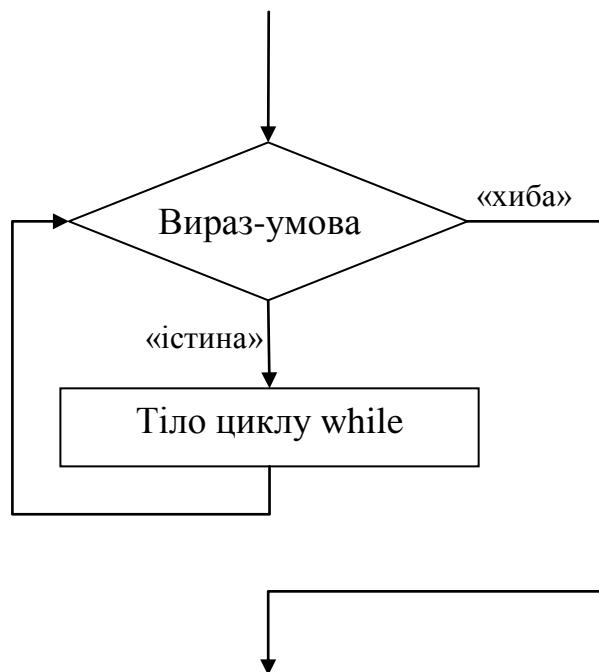


Рис. 4.10 — Блок-схема оператора циклу while

Враховуюче зазначене, фрагмент програмного коду, в якому використовується оператора циклу **for**:

```

Sum = 0;
for ( x = 1;     x <= (2*N-1);     x += 2 ) {
    Sum += x;

    if( x < (2*N-1) )
        printf("%u + ", x);
    else
        printf("%u", x);
}

```

Може бути переписаний із використанням оператора циклу while:

```

Sum = 0;
x = 1;
while ( x <= (2*N-1) ) {
    Sum += x;

    if( x < (2*N-1) )
        printf("%u + ", x);
    else
        printf("%u", x);

    x += 2;
}

```

Порівнюючи обидва варіанти запису, можна помітити як змінилося місце розташування виразу ініціалізація (`x = 1`) та виразу оновлення (`x += 2`) при зміні циклу `for` на цикл `while`.

4.6 Оператор циклу do/while

Оператор циклу `do/while` на відміну від операторів `for` та `while`, передбачає визначення результату виразу-умови не перед виконанням ітерації, а вже після виконання ітерації. Таким чином, оператор `do/while` — це цикл з пост-умовою. Це значить, що в циклі `do/while` спочатку виконуються всі оператори, які формують тіло циклу, і лише після цього буде виконуватися перевірка умови та буде визначатися чи потрібно переходити до наступної ітерації, або чи потрібно виходити із циклу і переходити до наступного оператора, що знаходиться після оператора `do/while`. Враховуючи, що визначення логічного результату у виразі-умові виконується після проходу по

циклу, то гарантовано в циклі do/while буде виконано щонайменше один прохід по тілу циклу (гарантовано буде виконана одна ітерація), і у випадку, якщо вираз-умова одразу ж при першому визначенні його результату буде мати логічну «хибу», виконається завершення циклу. Таку особливість функціонування оператора do/while треба брати до уваги при розробці та реалізації відповідних алгоритмів.

В загальному випадку оператор do/while має такий формат запису [5—7]:

```
do
{
    Оператори, які виконуються в циклі;
}
while ( вираз-умова );
```

Може застосовуватися наступний формат запису фігурних дужок, які призначені для формування блоку, що представляє собою тіло циклу:

```
do {
    Оператори, які виконуються в циклі;
} while ( вираз-умова );
```

Якщо в тілі циклу передбачається виконання лише одного оператора, то фігурні дужки можна не використовувати при записі оператора do/while:

```
do
    Один оператор, який виконується в циклі;
while ( вираз-умова );
```

Важливо пам'ятати, що на відміну від операторів for та while, в операторі циклу do/while після круглих дужок, в яких записується вираз-умова, необхідно обов'язково ставити символ «крапка з комою».

Якщо символ «крапка з комою» в операторі do/while після закриваючої круглої дужки пропущений, то це є синтаксичною помилкою, і на етапі компіляції програми компілятор видасть відповідне повідомлення про помилку.

Якщо ж символ «крапка з комою» буде поставлений після виразу умови в операторі циклу for або while, — то це не буде синтаксичною помилкою, але це може бути логічною помилкою, тобто такою помилкою, при якій формально

програма буде написана правильно, але виконуватися відповідний фрагмент коду буде іншим чином, ніж розраховував розробник. На це необхідно звертати увагу.

Блок-схема оператора циклу do/while показана на рис. 4.11 [4].

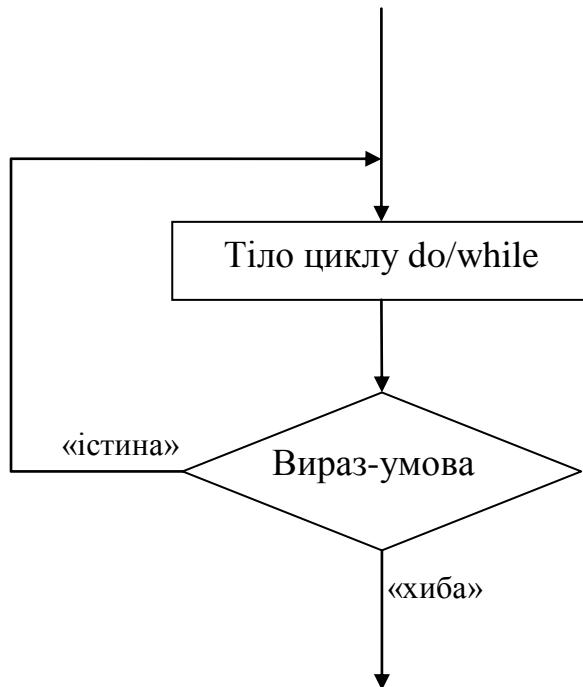


Рис. 4.11 — Блок-схема оператора циклу do/while

Вибір відповідного циклічного оператора — for, while або do/while, — часто продиктовано зручністю використання відповідного оператора циклу для реалізації відповідної частини алгоритму. Але часто фрагмент коду, в якому використовується певний оператор циклу, може бути переписаний таким чином, щоб використовувати інший оператор циклу, без зміни логіки роботи програми.

Наприклад, раніше розглянутий фрагмент коду, в якому відбувався підрахунок суми числового ряду, і який був реалізований за допомогою операторів циклу for та while, може бути переписаний із використанням оператора циклу do/while:

```
Sum = 0;  
x = 1;
```

```

do {
    Sum += x;

    if( x < (2*N-1) )
        printf("%u + ", x);
    else
        printf("%u", x);

    x += 2;
}while( x <= (2*N-1) );

```

Так само як умовні оператори вибору (if, if/else, switch) можуть містити відповідні вкладені оператори, так і оператори циклу for, while, do/while також можуть містити вкладені оператори циклу.

4.7 Оператори `continue` та `break` та їх використання в операторах циклу

В тілі операторів циклу можуть використовуватися оператори, які можуть впливати на виконання циклу. До таких операторів відносять `continue` та `break`.

Оператор `continue` дозволяє перервати виконання поточної ітерації та перейти до виконання наступної ітерації циклу (нової ітерації) [3—8].

Якщо оператор `continue` зустрічається в циклі `for`, то спочатку обчислюється вираз *оновлення*, потім виконується визначення результату *виразу-умови*, і якщо *вираз-умова* дорівнює логічній «істині», тоді відбувається перехід до виконання наступної ітерації.

При використанні оператора `continue` в циклі `while` або `do/while`, то перед тим як розпочинати нову ітерацію — виконується перевірка *виразу-умови*, і якщо результат *виразу-умови* — логічна «істина», тоді відбувається виконання чергової ітерації.

Оператор `break` використовується з метою переривання виконання оператора циклу, тобто дозволяє виконати вихід із циклу [3—8]. Часто оператор `break` використовується, щоб завершити виконання нескінченного

циклу. Приклади використання операторів продемонстровано в коді програми, яка представлена нижче. Програма реалізує алгоритм обрахунку добутку непарних додатних цілих значень, які користувач вводить з клавіатури, причому до обрахунку будуть братися тільки значення із діапазону 1...10. Якщо користувач буде вводити додатні парні значення — вони не будуть прийматися до обрахунку добутку. Як тільки користувач введе 0 або від'ємне число — процес введення даних завершиться і на екран в консольне вікно буде виведено обрахований результат. В програмі використовується нескінчений цикл — `while(1)`, що дозволяє трактувати вираз-умову в дужках як «істина», враховуючи, що у виразі-умові записана константа 1, то в процесі виконання програми дана умова не зміниться, і вона завжди буде істинною, тому в програмі передбачено вихід із циклу за допомогою використання оператора `break`, який викликається, коли користувач вводить з клавіатури значення, що менше або дорівнює 0. Якщо користувач вводить парне додатне значення або непарне значення, що більше 10, то оператор `continue` дозволяє пропустити залишок тіла циклу, де записані вирази по обрахунку добутку, і перейти на нову ітерацію. В залежності від значення змінної `count_number` буде відображатися відповідний результат.

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

int main()
{
    int result;           // result – зберігає результат добутку
    int value;            // value – поточне значення, яке вводить користувач
    int count_number;    // count_number – кількість чисел, які враховувалися при
                        // обрахунку результатау
    setlocale(LC_CTYPE, "Ukrainian");
    printf("Обраховується добуток непарних значень із діапазону 1...10. \n");
    printf("\nДодатні парні числа – ігноруються і не враховуються ");
    printf("при визначенні результатау.");
    printf("\nЯкщо ввести значення, що менше або дорівнює 0 – процес ");
    printf("\nвведення числових значень завершується, і виводиться ");
    printf("отриманий результат.\n\n");
```

```

result = 1;
count_number = 0;

while( 1 ) // Умова завжди істинна. Цикл нескінчений
{
    printf("Введіть ціле число: ");
    scanf("%d", &value);

    if( value <= 0 )
        break; // перервати виконання циклу. Вийти із циклу while
    else if ( (value%2 == 0) || (value > 10) )
        continue; // пропустити залишок тіла циклу, перейти на нову ітерацію

    result *= value;
    count_number++;
}

switch( count_number )
{
    case 0:
        printf("\n\n-----");
        printf("\nРезультат відсутній.\n");
        break;

    default:
        printf("\n\nРезультат: %d\n\n", result);
}
return 0;
}

```

На рис. 4.12 показаний результат роботи програми, коли користувач вводив значення, які не враховувалися при визначенні добутку.

```

Обраховується добуток непарних значень із діапазону 1...10.
Додатні парні числа - ігноруються і не враховуються при визначенні результату.
Якщо ввести значення, що менше або дорівнює 0 - процес
введення числових значень завершується, і виводиться отриманий результат.

Введіть ціле число: 2
Введіть ціле число: 4
Введіть ціле число: 15
Введіть ціле число: -3

-----
Результат відсутній.

Process returned 0 <0x0>   execution time : 19.773 s
Press any key to continue.

```

Рис. 4.12 — Результат виконання програми для випадку коли вхідні дані не містять значень допустимих умовою завдання

На рис. 4.13 приведений результат роботи програми, коли користувачем були введені вхідні дані серед яких були непарні цілі числа із діапазону 1...10, які враховувалися при визначенні результату.

```
Обраховується добуток непарних значень із діапазону 1...10.  
Додатні парні числа - ігноруються і не враховуються при визначенні результату.  
Якщо ввести значення, що менше або дорівнює 0 - процес  
введення числових значень завершується, і виводиться отриманий результат.  
Введіть ціле число: 2  
Введіть ціле число: 3  
Введіть ціле число: 4  
Введіть ціле число: 5  
Введіть ціле число: 6  
Введіть ціле число: 7  
Введіть ціле число: 8  
Введіть ціле число: 9  
Введіть ціле число: 10  
Введіть ціле число: 11  
Введіть ціле число: 12  
Введіть ціле число: 13  
Введіть ціле число: 14  
Введіть ціле число: 0  
  
Результат: 945
```

Рис. 4.13 — Результат виконання програми

4.8 Питання для самоконтролю

1. Поясніть яку кількість ітерацій буде виконано в наступному фрагменті коду:

```
while(25)  
    printf("*");
```

2. Поясніть як буде виконуватися приведений фрагмент коду:

```
i = 0;  
while(i<100) {  
    continue;  
    printf("*");  
    i++;  
}
```

3. Навіщо використовується оператор `break` при роботі із циклічними операторами?

4. Яка відмінність між операторами `break` та `continue` при роботі із циклами?

5. Що таке ітерація?

6. Скільки ітерацій виконається, при хибному виразі-умові циклу `do/while`?

7. Запропонуйте приклади застосування операторів вибору та операторів циклу.

5. ВКАЗІВНИКИ. ОСНОВИ РОБОТИ ІЗ ФУНКЦІЯМИ

До цього моменту при написанні програм використовувалися змінні, константи, а також функції стандартної бібліотеки.

Загалом, при використанні змінної необхідно було обрати тип даних для змінної, обрати ім'я змінної, а також присвоїти відповідне значення цій змінній. Для того щоб використовувати відповідне значення, яке зберігається в змінній в якомусь виразі чи операторі, необхідно було в такому виразі записати ім'я відповідної змінної.

В мові програмування С доступним є особливий тип даних, який призначений для зберігання адреси пам'яті. Такий елемент називається *вказівником*. Отже, *вказівники* призначені для зберігання адреси пам'яті, наприклад, адреси деякої змінної або елементу масиву тощо. Вказівники дуже широко використовуються в мові програмування С, наприклад, в якості параметрів функцій або в якості значення, що повертається функцією.

Ряд функцій стандартної бібліотеки потребують в якості параметра, що передається у функцію, адреси змінної, адреси масиву, адреси символального рядка та ін. Операція, яка дозволяє отримати адресу позначається знаком амперсанд — «&». Із цією операцією доводилося мати справу при використанні функції `scanf()` при введенні значень змінної з клавіатури. Функція `scanf()` отримувала в якості параметра адресу в пам'яті відповідної змінної, і записувала у відповідну комірку пам'яті те значення, яке вводилося користувачем з клавіатури. Тим самим, змінюючи значення змінної, приписуючи змінній значення, що було задане користувачем.

Таким чином, знаючи адресу змінної можна дістатися до значення змінної і, наприклад, змінити це значення на яке-небудь інше. Вказівники дають можливість використовувати цей потужний інструментарій при роботі із даними. Тому важливо розібратися із основними принципами застосування вказівників при вирішенні різноманітних задач обробки даних.

5.1 Особливості використання вказівників

Вказівник — це такий вид даних, який призначений для зберігання адреси пам'яті [6]. Пам'ять можна схематично представити у вигляді набору комірок. Кожна комірка має свій номер — цей номер може розглядатися як адреса. Розмір окремої такої комірки — 1 *Байт*. Таким чином, кожна комірка пам'яті в розміром 1 байт має свій номер (свою адресу). Для зберігання значень різних типів даних передбачено різний об'єм пам'яті. Наприклад, для значення типу `char` виділяється 1 байт пам'яті, так само як і для типу даних `unsigned char`. Для значення типу `int` виділяється 4 байти пам'яті, так само як і для значень типу `signed int` та `float`. Для значення типу `double` виділяється 8 байт пам'яті і т.д. Тобто, значення відповідного типу, яке в комп'ютері зберігається в двійковому коді, буде потребувати різного об'єму пам'яті для свого збереження, тобто різну кількість однобайтних комірок пам'яті. Наприклад, нехай в програмі використовуються наступні змінні (яким одразу при оголошенні було присвоєно певні значення):

```
char    a = -44;
double b = 90.5;
float  c = 0.125;
int    d = 27;
```

Схематичне розміщення в пам'яті відповідних змінних показано на рис. 5.1. Вказівники в програмі оголошуються відповідно до такого формату запису [5, 7]:

```
тип_даних * im'я_вказівника;
```

В наступному записі виконується оголошення відповідних вказівників, які будуть призначені для зберігання: адреси змінної типу `char`, адреси змінної типу `double`, адреси змінної типу `float` та адреси змінної типу `int`:

```
char * pa;
double * pb;
float  * pc;
int   * pd;
```

Тип даних змінної	<code>char</code>	<code>double</code>								<code>float</code>				<code>int</code>			
Ім'я змінної	<code>a</code>	<code>b</code>								<code>c</code>				<code>d</code>			
Значення змінної	<code>-44</code>	<code>90.5</code>								<code>0.125</code>				<code>27</code>			
Адреса в пам'яті	<code>7B</code>	<code>7C</code>	<code>7D</code>	<code>7E</code>	<code>7F</code>	<code>80</code>	<code>81</code>	<code>82</code>	<code>83</code>	<code>84</code>	<code>85</code>	<code>86</code>	<code>87</code>	<code>88</code>	<code>89</code>	<code>8A</code>	<code>8B</code>
Розмір	1 Байт	8 Байт								4 Байт				4 Байт			

Рис. 5.1 — Схематичне розміщення в пам'яті змінних різних типів

Якщо вказівники оголошенні, наприклад, в тілі функції `main()`, то по аналогії із звичайними змінними, вказівники представляють собою локальні змінні, які призначенні для зберігання адрес, і при оголошенні вони вважаються невизначенними, тобто невідомо чому вони дорівнюють. Тому часто перед тим, як працювати із вказівниками, їм необхідно присвоїти відповідне значення, тобто адресу.

Імена для вказівників повинні обиратися виходячи із тих же правил, які використовуються для вибору імен змінних. В наведеному вище прикладі вказівник `ra` буде призначатися для зберігання адреси, за якою зберігається в пам'яті значення типу даних `char`, тому можна говорити, що `ra` — це вказівник на `char`. Вказівник `rb` буде використовуватися для зберігання адреси, за якою зберігається в пам'яті значення типу даних `double`, отже, `rb` — це вказівник на `double`. Аналогічно, `rc` — це вказівник на `float`, а `pd` — це вказівник на `int`.

Для того, щоб присвоїти вказівнику значення адреси відповідної змінної, потрібно до змінної відповідного типу застосувати операцію «отримати адресу», при цьому тип змінної повинен узгоджуватися із типом вказівника.

Для приведеного прикладу, вказівник `ra` буде отримувати адресу змінної `a`, вказівник `pb` буде отримувати адресу змінної `b`, вказівник `pc` буде отримувати адресу змінної `c`, вказівник `pd` буде отримувати адресу змінної `d`:

```
ra = &a;
pb = &b;
pc = &c;
pd = &d;
```

Таким чином, значення вказівників (відповідно до схеми розміщення в пам'яті приведених змінних, що показана на рис. 5.1) будуть такими:

```
ra = 7B;
pb = 7C;
pc = 84;
pd = 88;
```

Для значень адрес використовують шістнадцяткову систему числення, тому в наведеному прикладі значення вказівників — це шістнадцяткові коди.

Схематичне розміщення в пам'яті змінних, з урахуванням використання відповідних вказівників, показано на рис. 5.2.

Тип даних змінної	<code>char</code>	<code>double</code>								<code>float</code>				<code>int</code>			
Ім'я змінної	<code>a</code>	<code>b</code>								<code>c</code>				<code>d</code>			
Значення Змінної	-44	90.5								0.125				27			
Адреса в пам'яті	7B	7C	7D	7E	7F	80	81	82	83	84	85	86	87	88	89	8A	8B
	↑	↑								↑				↑			
	ra	pb								pc				pd			

Рис. 5.2 — Вказівники

Таким чином, кожен вказівник буде зберігати значення адреси першого байту області пам'яті відповідного розміру (розмір залежить від типу даних).

Для того, щоб отримати доступ до значення, що зберігається у відповідній області пам'яті, адреса першого байту якої зберігається у відповідному вказівнику, необхідно до вказівника застосувати операцію «розіменування», яка позначається символом «*» (цю операцію можна ще назвати як «доступ до значення за адресою»). При цьому, застосовуючи до вказівника операцію розіменування, і тим самим отримуючи доступ до відповідного значення, потрібно пам'ятати про типи даних. Нижче показаний фрагмент коду, який дозволяє вивести в консольне вікно значення змінних *a*, *b*, *c*, *d*, використовуючи для цього відповідно вказівники *pa*, *pb*, *pc*, *pd*, яким попередньо були присвоєні адреси зазначених змінних:

```
printf("\n *pa = %d", *pa );
printf("\n *pb = %lf", *pb );
printf("\n *pc = %f", *pc );
printf("\n *pd = %d", *pd );
```

На рис. 5.3 показано результат виконання даного фрагменту коду, який дозволяє в консольному вікні відобразити відповідні значення.

На рис. 5.4 показано результат виведення значень відповідних змінних в консольне вікно, використовуючи для цього імена змінних:

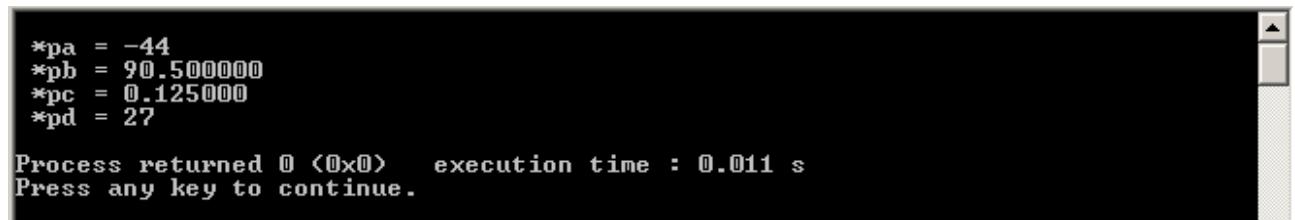
```
printf("\n a = %d", a );
printf("\n b = %lf", b );
printf("\n c = %f", c );
printf("\n d = %d", d );
```

Порівнюючи результат на рис. 5.3 та 5.4 можна помітити, що результати ідентичні.

Для того, щоб вивести в консольне вікно значення відповідних вказівників, тобто значення адрес відповідних змінних, що збережені у вказівниках *pa*, *pb*, *pc*, *pd*, — потрібно використовувати у функції *printf()* специфікатор формату *%p*:

```
printf("\n pa = %p", pa );
printf("\n pb = %p", pb );
printf("\n pc = %p", pc );
printf("\n pd = %p", pd );
```

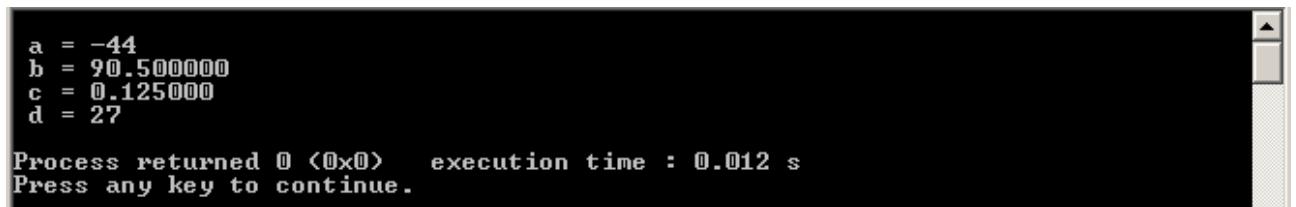
Приклад результату виконання фрагменту коду по виведенню значень вказівників на екран показано на рис. 5.5.



```
*pa = -44
*pb = 90.500000
*pc = 0.125000
*pd = 27

Process returned 0 <0x0>   execution time : 0.011 s
Press any key to continue.
```

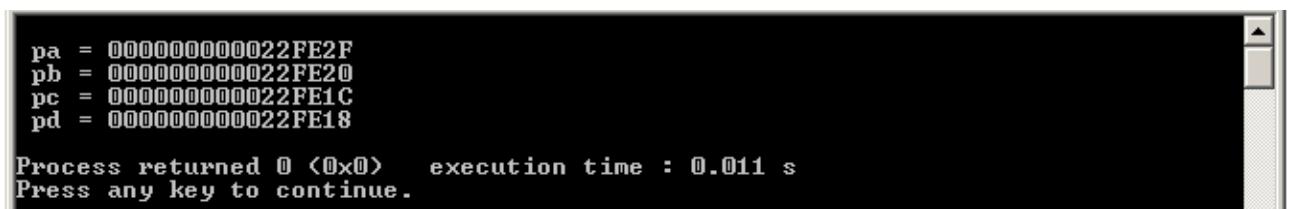
Рис. 5.3 — Використання вказівників для доступу до значень відповідних змінних та виведення цих значень в консольне вікно.



```
a = -44
b = 90.500000
c = 0.125000
d = 27

Process returned 0 <0x0>   execution time : 0.012 s
Press any key to continue.
```

Рис. 5.4 — Виведення в консольне вікно значень змінних різних типів даних



```
pa = 00000000000022FE2F
pb = 00000000000022FE20
pc = 00000000000022FE1C
pd = 00000000000022FE18

Process returned 0 <0x0>   execution time : 0.011 s
Press any key to continue.
```

Рис. 5.5 — Виведення в консольне вікно значень вказівників. Результати носять демонстраційний характер

Використовуючи вказівники, можна не тільки отримувати доступ до значень відповідних змінних, але і змінювати значення змінних. В приведеному нижче фрагменті коду значення всіх змінних встановлюються рівними 0, при

чому для цього використовуються не імена відповідних змінних, а їх адреси, що збережені у відповідних вказівниках. Щоб мати змогу за допомогою вказівника отримати доступ до значення змінної і мати можливість це значення змінити, використовується операція «розіменування» (або називають «роздресація» [7]), яка позначається символом «*». На рис. 5.6 виводяться значення змінних після виконання наступного фрагменту коду:

```
*pa = 0;  
*pb = 0;  
*pc = 0;  
*pd = 0;
```

```
a = 0  
b = 0.000000  
c = 0.000000  
d = 0  
Process returned 0 <0x0> execution time : 0.012 s  
Press any key to continue.
```

Рис. 5.6 — Значення змінних встановлюються рівними нулю за допомогою використання вказівників

В програмі можуть бути використані декілька вказівників, які можуть зберігати адресу однієї і тієї ж змінної. Таким чином, за допомогою кожного із таких вказівників можна отримати доступ до значення змінної, а також можна змінити значення змінної.

Важливо пам'ятати, що застосовувати операцію «розіменування» до неініціалізованого вказівника з метою запису деякого значення в область пам'яті, на яку вказує вказівник — заборонено (бо неініціалізований вказівник може мати (аналогічно до локальних змінних), якесь довільне значення адреси, або нульову адресу, і при намаганні записати за цією адресою деяке значення, може виникнути аварійне завершення програми). Тому перед тим як застосувати до вказівника операцію «*», необхідно вказівнику присвоїти адресу, наприклад, адресу змінної [3, 4].

При оголошенні вказівника йому можна присвоїти значення NULL (це може розглядатися як певний аналог присвоєнню деякій змінній значення 0,

щоб вона мала певне визначене значення перед тим, як їй буде присвоєно деяке значення в ході виконання програми на подальших кроках реалізованого алгоритму) [4].

Окремо можна виділити вказівник на `void` — це такий вказівник, який призначений для зберігання значення адреси. Розіменовувати безпосередньо вказівник на `void` — заборонено [3, 4]. Однак, використовуючи операцію приведення типів, можна використовувати вказівник на `void`, щоб дістатися до відповідного значення, яке зберігається в області пам'яті, адреса якої зберігається у вказівнику на `void`. Але при цьому треба брати до уваги, що приведення типів при розіменуванні вказівника на `void`, буде визначати яким чином буде виконуватися інтерпретація двійкового коду, що зберігається у відповідній області пам'яті, а також який розмір в байтах відповідної області пам'яті потрібно буде розглядати. Приклад використання вказівника на `void`, а також приклад застосування операції приведення типів при її застосуванні до вказівників, показано в програмному коді, що записаний нижче.

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

int main()
{
    float x = 4500.125;
    int y = 2;

    void * ptr = NULL;

    printf("\n x=%f", x);
    printf("\n y=%d", y);
    printf("\n ptr=%p", ptr);

    ptr = &x;

    printf("\n\n ptr=%p", ptr);
```

```

printf("\n\n *((float *)ptr)=%f", * ((float *)ptr) ) ;

printf("\n\n");

*((float *)ptr) = *((float *)ptr) * 2;

printf("\n x=%f", x);

y = *((int *)ptr);

printf("\n y=%d", y);

printf("\n\n");
return 0;
}

```

На початку програми виконується ініціалізація змінних *x* та *y*, які мають тип даних *float* та *int* відповідно. Змінній *x* присвоєно значення 4500.125, а змінній *y* присвоєно значення 2.

Також виконується ініціалізація вказівника на *void*, що має ім'я *ptr*, якому присвоєно значення *NULL*.

За допомогою наступного фрагменту коду виводяться в консольне вікно значення *x* та *y*, а також значення *ptr*:

```

printf("\n x=%f", x);
printf("\n y=%d", y);
printf("\n ptr=%p", ptr);

```

Приклад виведення відповідних значень в консольне вікно, показано на рис. 5.7.

```

x=4500.125000
y=2
ptr=0000000000000000

```

Рис. 5.7 — Виведення початкових значень змінних та вказівника

За допомогою оператора *ptr = &x;* вказівнику *ptr* присвоюється значення адреси змінної *x*, а за допомогою виклику функції:

```
printf("\n\n ptr=%p", ptr);
```

значення вказівника `ptr` (адреса змінної `x`) виводиться в консольне вікно, що показого на рис рис. 5.8.

```
ptr=0000000000022FE3C
```

Рис. 5.8 — Значення вказівника `ptr` (адреса змінної `x`)

В наступному рядку виконується операція приведення типів `(float *)ptr`, а після цього застосовується операція розіменування: `*((float *)ptr)` — таким чином за допомогою операції приведення типів, значення адреси, що збережене у вказівнику `ptr` буде розглядатися як адреса області пам'яті (розмір цієї області пам'яті буде братися 4 байти, оскільки для типу `float` виділено саме 4 байти пам'яті), за якою розміщується значення типу `float`, а операція розіменування дозволяє отримати доступ до вмісту відповідної області пам'яті, і виконати інтерпретацію двійкового коду, який там зберігається, як значення типу даних `float` і вивести це значення на екран як десяткове число (рис. 5.9). Це реалізовано з використанням виклику функції `printf()`:

```
printf("\n\n *((float *)ptr)=%f", * ((float *)ptr) );
```

```
*((float *)ptr)=4500.125000
```

Рис. 5.9 — Виведення значення змінної `x`, використовуючи вказівник `ptr`

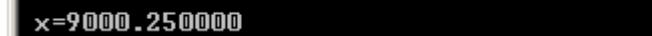
Таким чином, за допомогою вказівника на `void`, а також шляхом застосування операції приведення типів та операції розіменування вдалося отримати доступ до значення змінної `x` та вивести це значення в консольне вікно.

За допомогою вказівника `ptr` і виконання операцій приведення типів та розіменування відбувається зміна значення змінної `x` шляхом множення на 2 значення змінної `x` і збереження його за адресою змінної `x`:

```
* ((float *)ptr) = * ((float *)ptr) * 2;
```

За допомогою виклику функції `printf()` ввідбувається виведення значення змінної `x` в консольне вікно (рис. 5.10):

```
printf("\n x=%f", x);
```



```
x=9000.250000
```

Рис. 5.10 — Виведення значення змінної `x`, після її зміни, шляхом множення на 2, використовуючи для цього вказівник `ptr`

Після цього за допомогою наступного фрагменту коду:

```
y = *((int *)ptr);
```

змінній `y` присвоюється значення, але при цьому до вказівника `ptr` застосовується операція приведення типів `(int *)ptr`, що дозволяє виконати інтерпретацію адреси, яка зберігається у вказівнику, як адреси області пам'яті, за якою зберігається значення типу `int`. Тип `int` так само як і тип даних `float` — має розмір 4 байти. Виконання операції приведення типів `(int *)ptr` дозволяє розглядати відповідну область пам'яті розміром 4 байти як таку, за якою зберігається значення `int`, тим самим двійковий код, який записаний в даній області пам'яті, буде інтерпретуватися як двійковий код типу даних `int`. Застосувавши операцію розіменування: `*((int *)ptr)` відповідне числове значення (яке зберігається в області пам'яті, адреса якої зберігається в `ptr`) буде присвоєно змінній `y`, після чого значення змінної виводиться на екран в консольне вікно за допомогою функції `printf()` (рис. 5.11):

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



```
y=1175232768
```

Рис. 5.11 — Виведення значення змінної `y`

Таким чином, двійковий код (який зберігався в області пам'яті розміром 4 байти, адреса якої зберігалася у вказівнику `ptr`), і має таку послідовність бітів:

```
0100 0110 0000 1100 1010 0001 0000 0000
```

В одному випадку інтерпретувався як значення типу `float` ($x = 9000.25$), а в другому випадку, інтерпретувався як значення типу `int` ($y = 1\,175\,232\,768$).

При оголошенні декількох вказівників одного типу, при записі оголошення в один рядок, важливо бути уважним і не забувати поряд з іменем вказівника записувати символ «`*`». Наприклад, в наступному фрагменті коду оголошуються 3 вказівники на `double`, що мають імена `p1`, `p2`, `p3`:

```
double * p1, * p2, * p3;
```

Символ «`*`» може записуватися безпосередньо біля імені вказівника, без пробілу:

```
double *p1, *p2, *p3;
```

Якщо при оголошенні декількох вказівників представленим способом, забути записати символ «`*`» поряд із іменем, як в прикладі нижче:

```
double * p1, p2, p3;
```

то `p1` — це вказівник на `double` (призначений для зберігання адреси), а `p2` та `p3` — це змінні типу `double` (призначенні для зберігання дійсних чисел).

При оголошенні вказівників може використовуватися службове слово **const** (в деяких джерела `const` називають *специфікатор*, а в інших джерелах — називають *модифікатором*).

З урахуванням наявності чи відсутності службового слова `const` при оголошенні вказівника, можливі наступні варіанти [4, 7]:

1. **Неконстантний вказівник на неконстантні дані.** Службове слово **const** не використовується в оголошенні вказівника, наприклад:

```
double * p1;
```

Вказівник `p1` — неконстантний вказівник на неконстантні дані. Тобто, значення вказівника може змінюватися в програмі (вказівнику можуть присвоюватися різні адреси), і за допомогою вказівника шляхом застосування операції розіменування може змінюватися значення (наприклад, значення деякої змінної), що розміщується за адресою, яка зберігається у вказівнику.

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

int main()
{
    const double PI = 3.14;      // константа Пі
    double radius = 1.0;        // змінна, що зберігає радіус кола
    double length;             // змінна для зберігання значення довжини кола

    double * p1;               // p1 - неконстантний вказівник на неконстантні дані

    p1 = &radius;              // p1 отримує адресу змінної radius
    *p1 = *p1 * 10;            // Значення змінної radius збільшується в 10 разів

    p1 = &length;              // p1 отримує адресу змінної length
    *p1 = 2*PI*radius;         // Значення змінної length зберігає
                               // обчислену довжину кола

    printf("\n\nR = %.1lf", radius); // R = 10.0
    printf("\nL = %.2lf", length);  // L = 62.80

    return 0;
}
```

2. Неконстантний вказівник на константні дані. Службове слово **const** записується перед іменем типу даних:

```
const double * p1;
```

Вказівник `p1` — неконстантний вказівник на *константні* дані. Тобто, значення вказівника може змінюватися в програмі (вказівнику можуть присвоюватися різні адреси), але ЗАБОРОНЕНО за допомогою вказівника шляхом застосування операції розіменування змінювати значення (змінної чи тим паче константи), адреса якого зберігається у вказівнику:

```

const double PI = 3.14; // константа Пі
double radius = 1.0; // змінна, що зберігає радіус кола
double length; // змінна для зберігання значення довжини кола

const double * p1; // неконстантний вказівник на КОНСТАНТНІ дані

p1 = &radius; // p1 отримує адресу змінної radius
length = 2 * PI * *p1; // length зберігає обчислену довжину кола

radius = 10.0;

p1 = &length; // p1 отримує адресу змінної length
*p1 = 2*PI*radius; // !!!! ПОМИЛКА
// За допомогою вказівника намагаємося
// змінити значення змінної length

```

3. Константний вказівник на неконстантні дані. Службове слово **const** записується перед іменем вказівника, причому вказівнику потрібно одразу присвоїти значення при оголошенні вказівника (тобто виконати ініціалізацію вказівника):

```

double radius = 1.0; // змінна

double * const p1 = &radius; // константний вказівник

```

Вказівник **p1** — **константний** вказівник на неконстантні дані. Тобто, значення вказівника ЗАБОРОНЕНО змінювати в програмі (вказівнику треба надати значення адреси при його оголошенні — тобто виконати ініціалізацію вказівника), але дозволено за допомогою вказівника шляхом застосування операції розіменування змінювати значення змінної, адреса якої зберігається у вказівнику:

```

const double PI = 3.14; // константа Пі

double length; // змінна для зберігання значення довжини кола

double radius = 1.0; // змінна, що зберігає радіус кола

double * const p1 = &radius; // КОНСТАНТНИЙ вказівник на НЕКОНСТАНТНІ дані
// p1 отримує адресу змінної radius

*p1 = *p1 * 10; // Значення змінної radius збільшується 10 разів

```

```

length = 2 * PI * *p1; // length зберігає обчислену довжину кола

p1 = &length; // !!!! ПОМИЛКА
// Намагаємося змінити значення константного
// вказівника шляхом присвоєння йому адреси
// змінної length

```

4. Константний вказівник на константні дані. Службове слово **const** записується перед типом даних та перед іменем вказівника, причому вказівнику потрібно присвоїти значення при його оголошенні (тобто виконати ініціалізацію вказівника):

```

double radius = 1.0; // змінна

const double * const p1 = &radius; // КОНСТАНТНИЙ вказівник
// на КОНСТАНТНІ дані

```

Вказівник **p1** — *константний* вказівник на *константні* дані. Тобто, значення вказівника ЗАБОРОНЕНО змінювати в програмі (вказівнику потрібно надавати значення при його оголошенні — тобто виконувати одразу ініціалізацію вказівника), а також ЗАБОРОНЕНО за допомогою вказівника шляхом застосування операції розіменування змінювати значення, адреса якого зберігається у вказівнику:

```

const double PI = 3.14; // константа Пі

double length; // змінна для зберігання значення довжини кола

double radius = 1.0; // змінна, що зберігає радіус кола

const double * const p1 = &radius; // p1 - КОНСТАНТНИЙ вказівник на
// КОНСТАНТНІ дані
// p1 отримує адресу змінної radius
// ЗАБОРОНЕНО змінювати
// значення вказівника (тобто
// заборонено присвоювати вказівнику
// якусь іншу адресу, ніж яка була
// задана при ініціалізації)
// ЗАБОРОНЕНО за допомогою вказівника

```

```

        // змінювати значення змінної radius

length = 2 * PI * *p1; // length зберігає обчислену довжину кола

*p1 = *p1 * 10; // !!!! ПОМИЛКА
// За допомогою вказівника намагаємося
// змінити значення змінної radius

p1 = &length; // !!!! ПОМИЛКА
// Намагаємося змінити значення константного
// вказівника шляхом присвоєння йому адреси
// змінної length

```

Вказівники широко використовуються в мові програмування С. В наведених вище прикладах показано як вказівник використовувався у арифметичних виразах, в яких за допомогою операції розіменування, що застосовувалася до вказівника, відбувався доступ до значення змінної, а також змінна значення.

Нижче приводиться простий демонстраційний приклад програми, в якій знаходиться сума трьох значень, які зберігаються у відповідних змінних, але при цьому в безпосередніх обрахунках використовуються не імена змінних, а відповідні вказівники до яких застосовується операція розіменування:

```

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

int main()
{
    int a, b, c, sum;
    int *p1, *p2, *p3, *p4;

    p1 = &a;
    p2 = &b;
    p3 = &c;
    p4 = &sum;

    *p4 = 0; // змінній sum присвоюється значення 0

```

```

setlocale(LC_CTYPE, "Ukrainian");

printf("a=");
scanf("%d", p1); // p1 - зберігає адресу змінної a
// при введенні значення змінної a
// з клавіатури, функція scanf() потребує
// адресу змінної. Враховуючи, що ця адреса
// збережена у вказівнику p1, тому
// функції scanf() можна передати цю адресу
// використовуючи вказівник p1.
// Тобто записаний варіант аналогічний
// якби було записано scanf("%d", &a);

printf("b=");
scanf("%d", p2); // scanf("%d", &b);

printf("c=");
scanf("%d", p3); // scanf("%d", &c);

*p4 = *p1 + *p2 + *p3; // sum = a + b + c

printf("\n\nСума=%d", *p4); // printf("\n\nСума=%d", sum);

if (*p4 > 0) // if (sum > 0)
    printf("\nРезультат додатний");
else if (*p4 < 0) // if (sum < 0)
    printf("\nРезультат від'ємний");
else
    printf("\nРезультат дорівнює нулю");

printf("\n"); // перехід на новий рядок

return 0;
}

```

Наведений приклад ілюструє деякі особливості застосування вказівників. Більш широко вказівники використовуються при роботі із функціями, коли функція отримує адреси відповідних змінних з метою мати можливість отримувати доступ до значень змінних, а також, в разі необхідності, змінювати значення змінних безпосередньо в тілі відповідної функції.

5.2 Вказівник на вказівник

Вказівник використовується, щоб зберігати адресу, наприклад, адресу змінної деякого типу даних. Разом з тим, значення вказівника також зберігається в пам'яті, яка має свою деяку адресу. Таким чином, може виникати необхідність зберігати адресу вказівника. Для цього використовується вказівник на вказівник [6].

Нехай задано наступні змінні:

```
int a = 25, b = 66;
```

Змінні мають тип даних `int`, а також змінним надані початкові значення.

Крім змінних, нехай задано два вказівники на `int`:

```
int *ptr1=NULL, *ptr2=NULL;
```

Вказівникам на `int`, які мають імена `ptr1` та `ptr2`, присвоюється значення `NULL`.

Крім двох вказівників на `int`, також задається вказівник на вказівник:

```
int **pp;
```

Вказівник на вказівник `pp` призначений для зберігання адреси вказівника на `int`. В данному прикладі, вказівник `pp` може зберігати адресу вказівника `ptr1` або `ptr2`. Вказівник `pp` також розміщується в пам'яті за певною адресою. Вказівник `pp` лише оголошений, але не ініціалізований, тому його значення не відомо. Таким чином, схематично розміщення в пам'яті відповідних змінних та вказівників, а також початкові їх значення, показано на рис 5.12, де над прямокутниками вказані імена змінних та вказівників; в прямокутниках записані відповідні початкові значення; а під прямокутниками записано адреси відповідних змінних та вказівників в пам'яті.

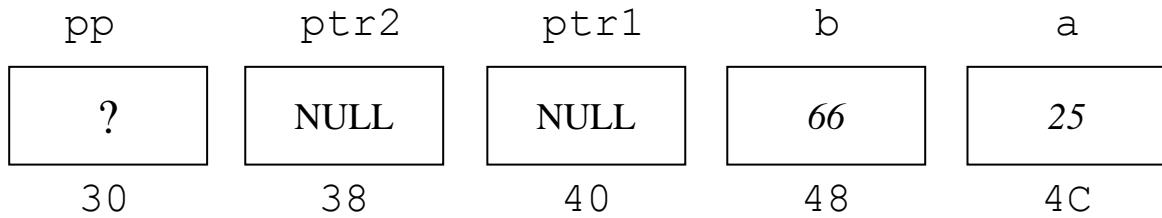


Рис. 5.12 — Схематичне розміщення в пам'яті змінних та вказівників

Нижче виконується присвоєння вказівникам ptr1 та ptr2 адреси змінної а та b відповідно:

```
ptr1 = &a;
ptr2 = &b;
```

При цьому значення вказівників відобразимо схематично на рис. 5.13, де вказівник ptr1 зберігає адресу змінної а (ptr1 вказує на змінну а), а вказівник ptr2 зберігає адресу змінної b (ptr2 вказує на змінну b).

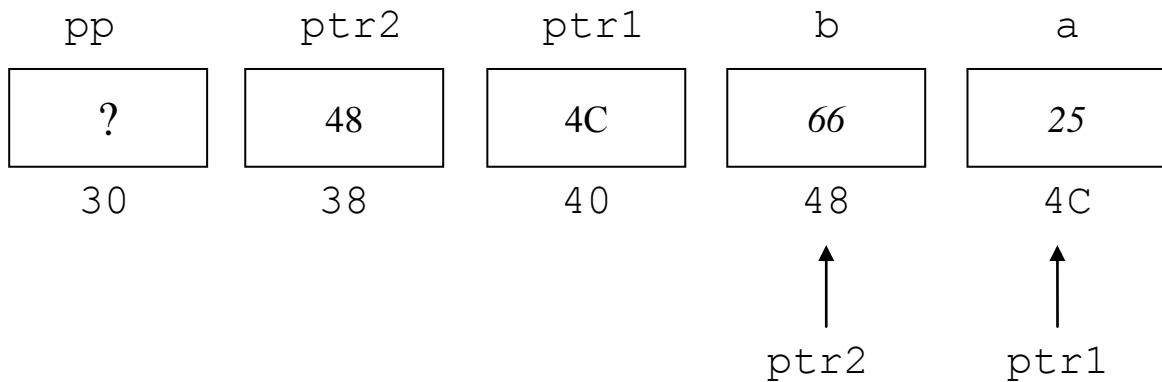


Рис. 5.13 — Вказівники зберігають адреси відповідних змінних

Щоб отримати доступ до значення змінної а, використовуючи для цього вказівник ptr1, необхідно виконати розіменування вказівника ptr1, а щоб отримати доступ до значення змінної b, використовуючи для цього вказівник ptr2, необхідно виконати розіменування вказівника ptr2. Наприклад, наступний запис дозволяє змінити значення змінних а та b, збільшивши значення змінної а на 100, а значення змінної b збільшивши в 10 разів:

```
*ptr1 = *ptr1 + 100;
*ptr2 = *ptr2 * 10;
```

Розміщення значень в пам'яті відповідних змінних після виконання приведених вище дій, показано на рис. 5.14.

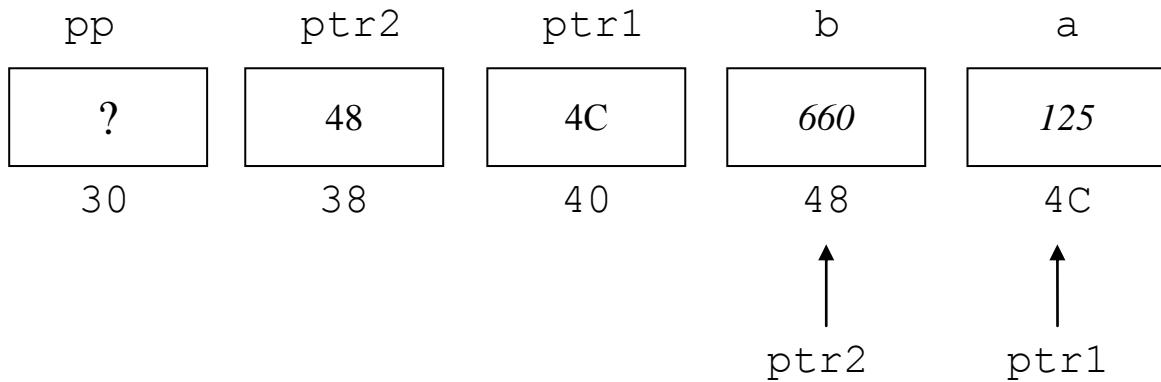


Рис. 5.14 — Результат зміни значень змінних при використанні вказівників

Вказівник pp може зберігати адресу вказівника на int. Нижче показано, що вказівнику pp присвоєна адреса вказівника ptr2:

```
pp = &ptr2;
```

Тепер вказівник pp зберігає адресу вказівника ptr2 (pp вказує на ptr2), схематично це показано на рис. 5.15.

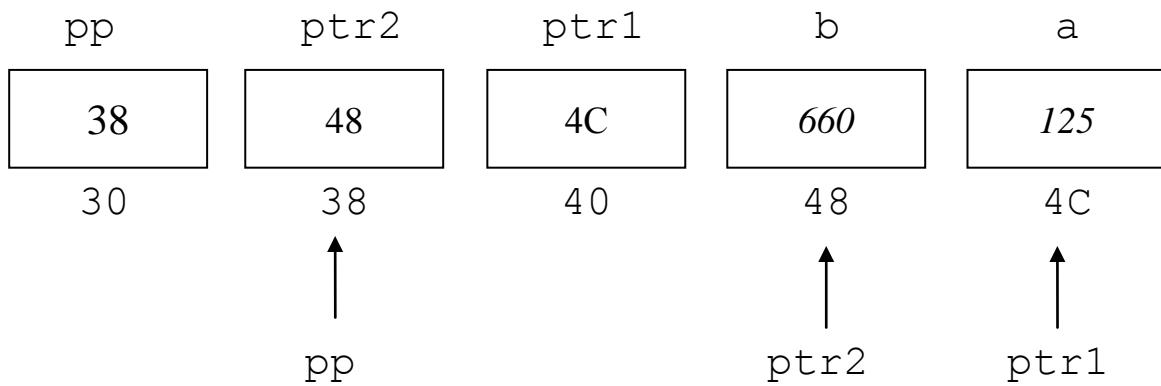


Рис. 5.15 — Схематичне розміщення в пам'яті змінних та вказівників

Якщо виконувати розіменування вказівника `pp` (застосувавши до нього операцію «`*`»), то це дасть можливість отримати значення вказівника `ptr2`, яке дорівнює адресі змінної `b` (рис. 5.16)

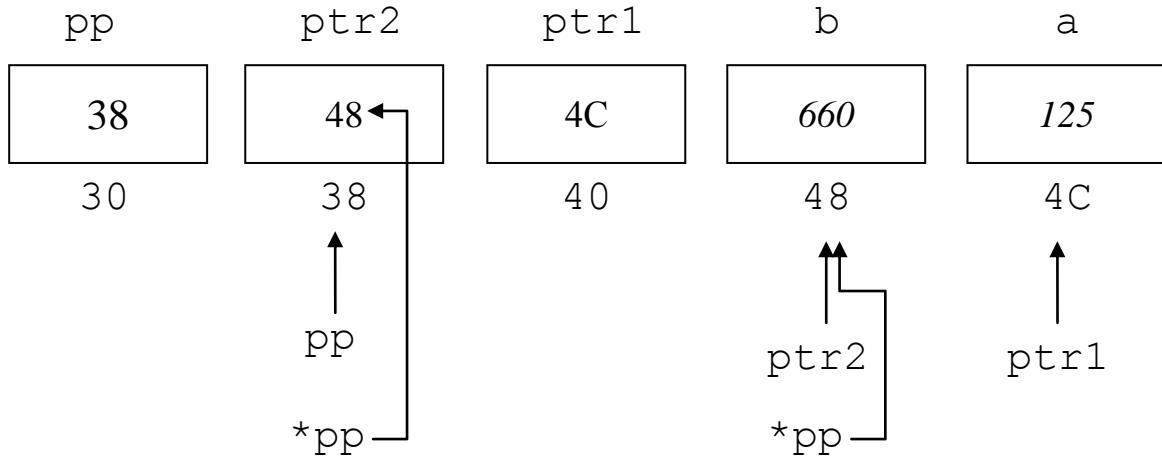


Рис. 5.16 — Розіменування вказівника на вказівник

Якщо до вказівника `pp` двічі застосувати операцію «`*`», то це дасть змогу дістатися до значення змінної `b`. В наступному рядку змінній `b` присвоюється значення 0 (рис. 5.17), при цьому для виконання таких дій використовується вказівник `pp`:

`**pp = 0;`

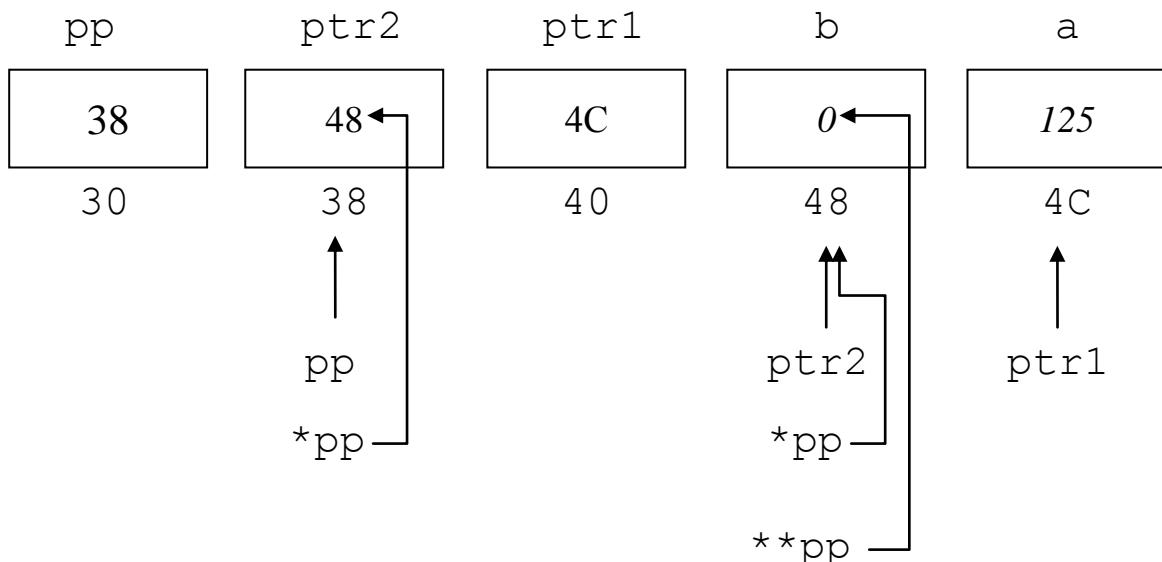


Рис. 5.17 — Зміна значення змінної при використанні вказівника на вказівник

Якщо до вказівника `pp` застосувати операцію розіменування один раз (записавши одну «`*`»), то це дасть можливість отримати доступ до значення вказівника `ptr2`, — можна, наприклад, використати значення вказівника `ptr2` або ж змінити це значення в залежності від необхідності виконання відповідних дій. Наприклад, наступний рядок, що показаний нижче, дозволяє змінити значення вказівника `ptr2`, присвоївши йому значення адреси змінної `a` (яка зберігається у вказівнику `ptr1`), причому для зміни значення `ptr2` використовується вказівник `pp`. Тепер обидва вказівники `ptr1` та `ptr2` зберігають адресу змінної `a` (`ptr1` та `ptr2` вказують на змінну `a`) (рис. 5.18):

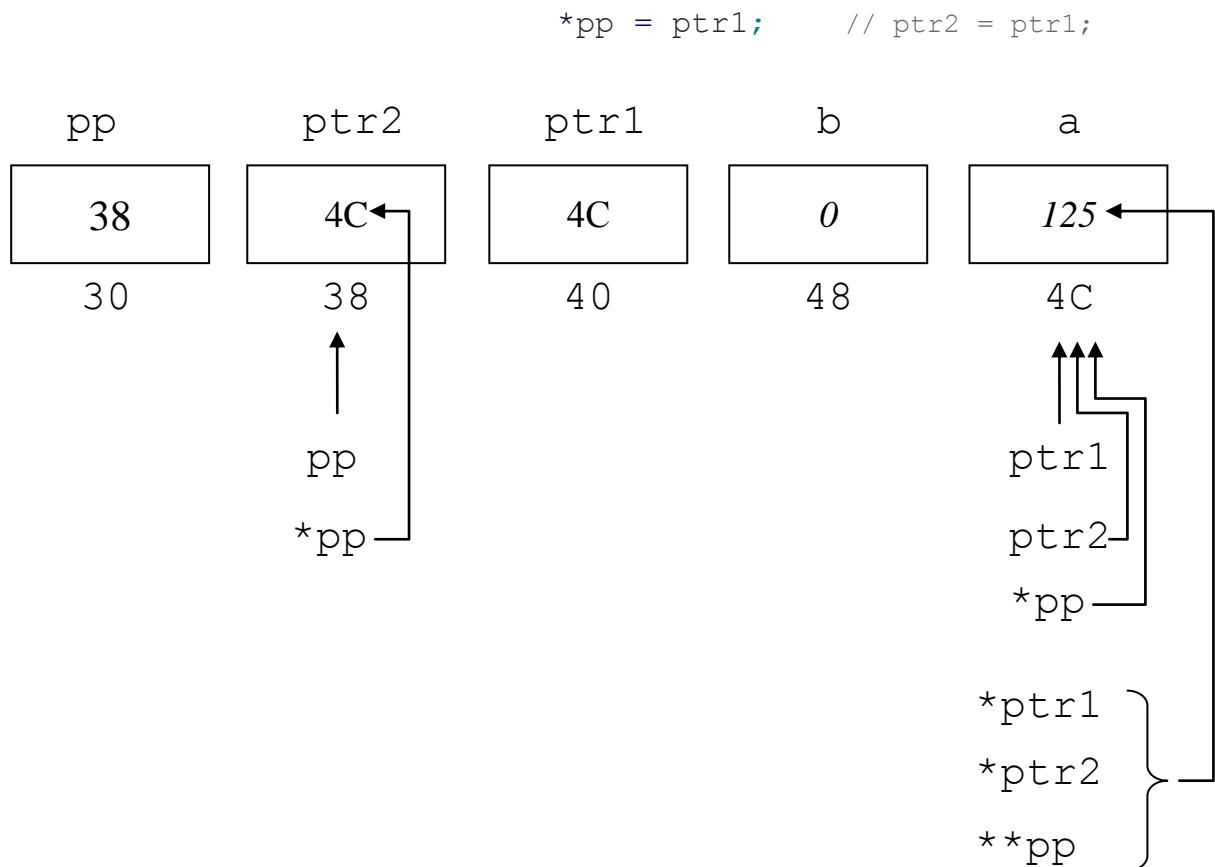


Рис. 5.18 — Вказівники `ptr1` та `ptr2` зберігають адресу змінної `a`

Таким чином, тепер до значення змінної `a` можна дістатися скориставшись вказівниками `ptr1`, `ptr2` та `pp`:

```

*ptr1 = *ptr1 / 25;      // a = a / 25;      a = 125/25 = 5
*ptr2 = *ptr1 + *ptr2;    // a = a + a;      a = 5+5 = 10
**pp = **pp + 4;         // a = a + 4;      a = 10 + 4 = 14

```

Схематичне розміщення відповідних значень в пам'яті після виконання вказаних дій, показано на рис. 5.19.

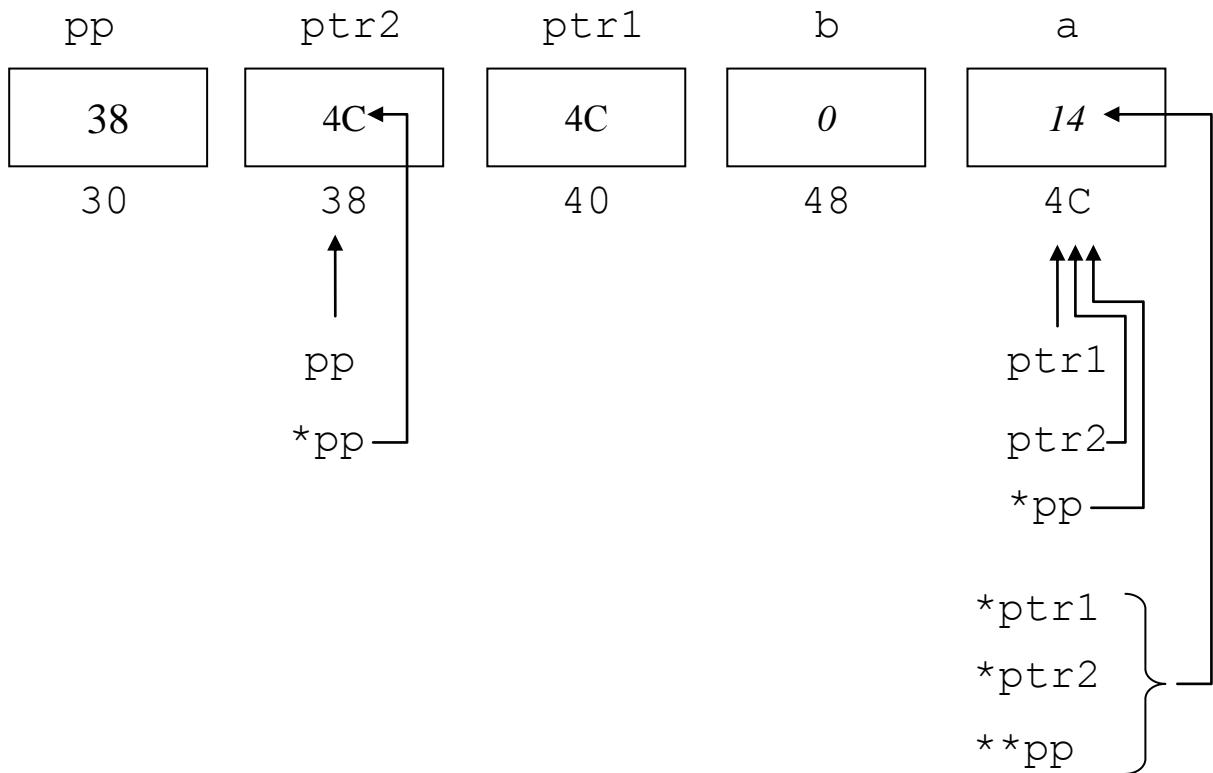


Рис. 5.19 — Доступ до значення змінної a за допомогою вказівників

При використанні вказівників потрібно бути дуже уважним. Необхідно завжди пам'ятати, що забороняється виконувати розіменування неініціалізованого вказівника [3, 4].

Вказівники часто доводиться використовувати не тільки при роботі із змінними, але також і при виконанні дій над масивами, символьними рядками та ін. Крім того, вказівники широко застосовуються при написанні функцій.

5.3 Основи роботи із функціями

Крім функцій стандартної бібліотеки часто доводиться створювати власні функції, які повинні реалізовувати відповідні алгоритми, або містити відповідний набір команд, які можуть бути написані із використанням функцій стандартної бібліотеки, а також сторонніх бібліотек. Загалом, в мові С функція представляє собою певну одиницю програмного коду, яка повинна вирішувати деяку задачу. Функції можна розділити на функції, що повертають результат, та функції, що не повертають результат (функції типу **void**) (рис. 5.20).

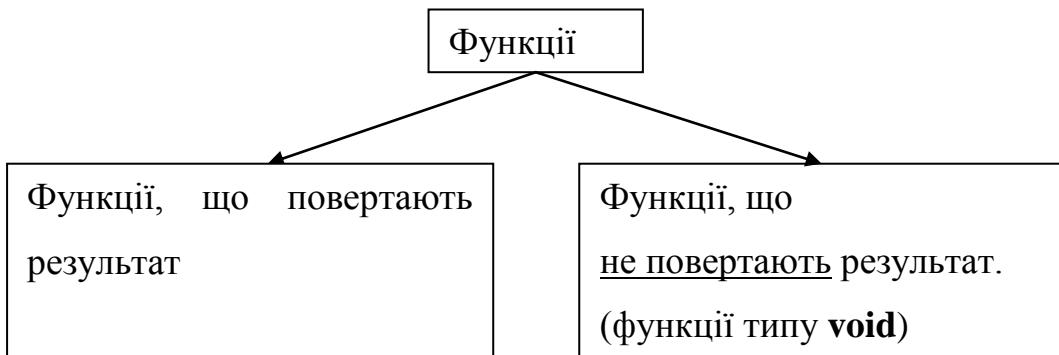


Рис. 5.20 — Види функцій

Крім того, функції можуть бути розділені на функції із параметрами та функції без параметрів.

Функції без параметрів не потребують передачу у функцію яких-небудь параметрів.

Функції з параметрами — потребують передачу у функцію параметрів, причому функції з параметрами можуть бути розділені на дві категорії в залежності від механізму передачі параметрів. Можна виділити наступні варіанти передачі параметрів у функцію [7]:

- 1) передача параметрів за значенням;
- 2) передача параметрів за посиланням (в якості параметрів виступають вказівники, тобто, адреси).

На рис. 5.21 схематично показана спрощена класифікація функцій, які будуть розглядатися нижче.

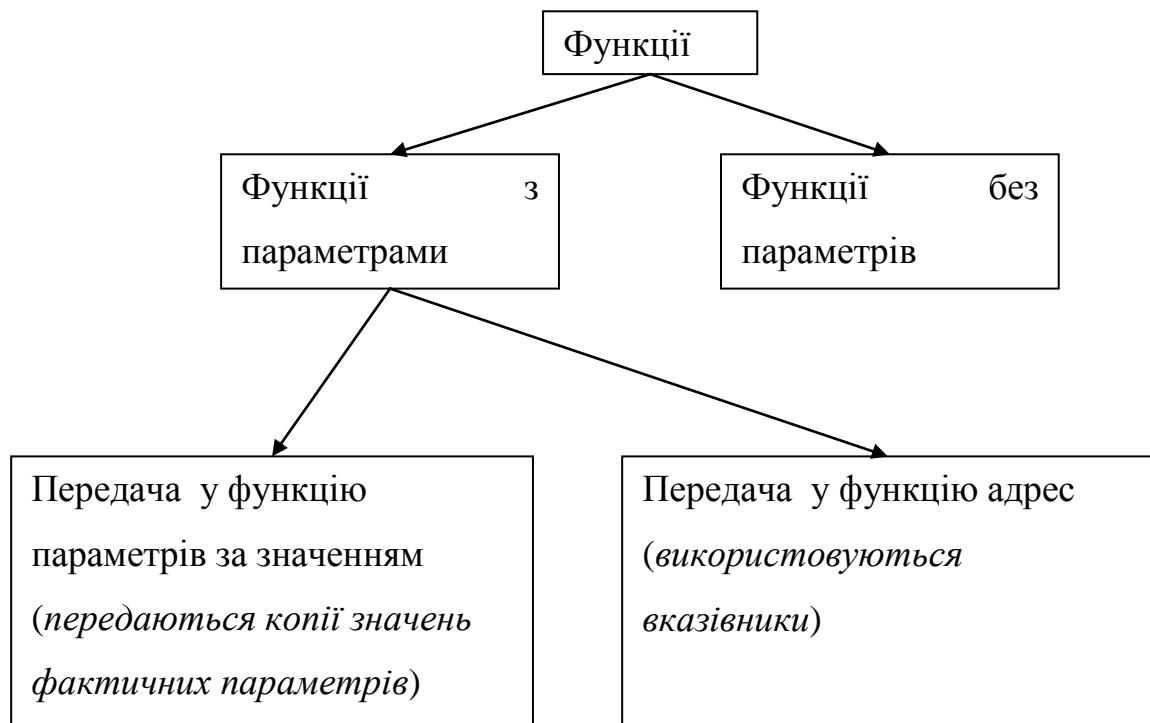


Рис. 5.21 — Спрощена класифікація функцій

Використання функцій дозволяє суттєво спростити виконання поставленого завдання, яке реалізується на мові програмування С. Головна задача, рішення якої потребує написання програми на мові С, може бути розділена на ряд підзадач, кожна з яких може розглядатися як логічно завершена послідовність дій, що вирішує певне завдання, і реалізується у вигляді окремої функції. Таким чином, вся програма в кінцевому варіанті може бути реалізована у вигляді виклику відповідних функцій, де кожна із функцій реалізує свою частину основної задачі.

Відповідно до представленої структури програми в розділі 3.1, при написанні власних функцій в програмі, необхідно спершу записати прототип функції.

Прототип функції записується відповідно до такого формату [4, 8]:

тип_даных_результату ім'я_функції (перелік_типів_даных_параметрів);

Якщо функція не повертає результат, то ***тип_даных_результату*** буде **void**.

Якщо функція не отримає параметрів, то ***перелік_типів_даных_параметрів*** буде містити службове слово **void**.

При записі ***переліку_типів_даных_параметрів*** також можна записувати крім типів даних ще й імена параметрів. Імена параметрів в прототипі функції ігноруються компілятором, тому імена параметрів в прототипі можна не вказувати. Однак часто імена параметрів все ж таки записують в прототипі функції, і головна мета цього — це інформування інших розробників, які будуть користуватися програмним кодом відповідних функцій, щодо призначення параметрів. Гарно обране ім'я, яке в своїй назві містить призначення параметра, дозволяє суттєво поліпшити написання та налагодження програми.

Імена функцій та імена параметрів — це ідентифікатори, тому при виборі імен потрібно притримуватися відповідних правил та вимог, які висуваються до складання ідентифікаторів та були визначені в розділі 3.

Нижче наведено приклади прототипів деяких довільних функцій:

```
void printHeader(void);
void showResult( float amount );
int getNumber( void );
double areaCircle( double radius );
void initCoef(int * plength, int * pwidth, int * pheight );
```

Виходячи із прототипів функцій можна отримати деяку важливу інформацію про функцію:

1. `void printHeader(void);` — функція має ім'я `printHeader`, що повинно говорити, що дана функція призначена для виведення деякого заголовку. В переліку параметрів (в круглих дужках) записано службове слово `void`, яке показує, що функція `printHeader` не отримує жодних параметрів. Службове

слово `void` також вказано як тип даних результату функції (записується перед іменем функції), тобто функція не повертає ніякого значення.

2. `void showResult(float amount);` — функція має ім'я `showResult`, що дає зрозуміти, що функція повинна відображати деякий результат на екрані монітору. В переліку параметрів вказано, що функція отримує один параметр типу даних `float`. Незважаючи на те, що імена параметрів в прототипі ігноруються компілятором, однак в даному випадку ім'я параметру вказано — виходячи із імені параметра (`amount`) можна зрозуміти, що функція отримує в якості параметру, наприклад, визначену суму. Параметр у функцію передається за значенням.

3. `int getNumber(void);` — функція має ім'я `getNumber` і виходячи із імені функції зрозуміло, що дана функція може призначатися для зчитування деякого числового значення. Функція не отримує жодного параметру, оскільки в переліку параметрів записано `void`. Функція повертає результат типу даних `int`, тобто деяке ціле знакове число.

4. `double areaCircle(double radius);` — функція має ім'я `areaCircle`, це дає зрозуміти, що функція призначена для обрахунку площі круга. Функція отримує один параметр типу даних `double`, а за рахунок вказування імені цього параметра (`radius`) стає зрозумілим, що функція потребує передачу радіуса круга. Тип даних результата, що функція повертає, — `double`, і очевидно, що функція буде повертати обраховане нею значення площі круга. Параметр у функцію передається за значенням.

5. `void initCoef(int * plength, int * pwidth, int * pheight);` — функція має ім'я `initCoef`, по імені функції зрозуміло, що дана функція може використовуватися для того, щоб виконувати ініціалізацію значень деяких параметрів. Функція має тип даних `void`, тобто функція не повертає результат. Функція отримує 3 параметри, причому ці параметри — це вказівники на `int`, тобто адреси, наприклад, деяких змінних. Імена параметрів `plength`, `pwidth`, `pheight` дають зрозуміти, що на відповідних позиціях повинні міститися

вказівники на змінні, які призначені для зберігання довжини (`plength`), ширини (`pwidth`) та висоти (`pheight`), літера «`p`» на початку імені додатково дає зрозуміти, що в якості параметрів очікуються саме адреси (вказівники), (англійською мовою вказівник — *pointer*). Параметри у функцію передаються за посиланням

Інформативні імена для функцій та параметрів — дозволяє суттєво поліпшити аналіз тексту програми та відповідає гарному стилю написання програмного коду.

Функції викликаються в програмі, шляхом запису імені функції, круглих дужок та переліку фактичних параметрів, або якщо функція параметрів не отримує, — то круглі дужки залишаються пустими. Виклик функції в загальному випадку може бути записаний так [4, 7, 8]:

ім'я_функції (*перелік_фактичних_параметрів*);

перелік_фактичних_параметрів — це ті параметри (аргументи), які записують при виклику функції, — це можуть бути імена змінних, що зберігають, наприклад, деякі значення; імена констант, арифметичні чи логічні вирази тощо. Наприклад, виклик функцій, прототипи яких були записані вище, могли бути такими (в наступних записах вважається, що відповідні змінні, які записані в наступному фрагменті коду були заздалегідь оголошені, а приведений виклик функцій носить демонстраційний характер):

```
printHeader();  
showResult( sum );  
x = getNumber();  
S = areaCircle( R );  
getCoef( &L, &H, &W );
```

Крім запису прототипу функції, а також виклику функції в певному місці програми, необхідно описати функції, тобто для кожної функції потрібно записати набір операторів, що формують тіло функції, і які повинні виконуватися, коли функція викликається. Опис функції починається із

заголовку, після якого між відкриваючою та закриваючою фігурними дужками записується тіло функції. В заголовку вказується **тип_даных_результату**, **ім'я функції** та **перелік_формальних_параметрів**. Перелік формальних параметрів містить перелік, в якому записується тип даних параметру та ім'я параметру. Вказані імена будуть використовуватися в тілі даної функції як локальні змінні. Важливо, щоб прототип функції та заголовок функції повністю узгоджувалися.

Якщо функція повертає результат, то повинен також бути записаний оператор **return** поряд з яким записується значення, що функція повертає — це може бути деяка константа, як наприклад, в функції **main()** — **return 0**; або це може бути ім'я змінної, що зберігає значення результату, який повертається функцією, наприклад, **return sum;** де **sum** — значення змінної, що зберігає деякий результат, який обраховується в тілі функції і повертається в точку виклику функції; або це може бути деякий вираз: **return sqrt(D);** тощо. Важливо при цьому, пам'ятати, що тип даних результату, який повертає функція, визначається на етапі написання прототипу. Оператор **return** завершує виконання функції і повертає відповідне значення в точку виклику [3—8]. Таким чином, після спрацювання оператора **return** всі інші оператори, які можуть міститися нижче в коді в тілі функції, — не виконуються.

Якщо описується функція типу **void**, то ця функція не повертає результат. Функція виконується поки не досягне завершення тіла функції або ж поки не буде виконаний оператор **return**. Для функцій типу **void** після оператора **return** не записується ніяких параметрів (оскільки функція типу **void** не повертає ніяких значень). Оператор **return** може викликатися, щоб завершувати виконання функції раніше, ніж буде досягнуто кінця тіла функції, наприклад, оператор **return** може записуватися в умовному операторів **if** і у випадку істинності відповідного виразу-умови оператора **if**, за допомогою оператора **return** може завершуватися виконання функції типу **void** [3—8].

В загальному випадку опис функції записується відповідно до такого формату [4, 5, 8]:

тип_даних_результату ім'я_функції (перелік формальних параметрів)

{

Тіло функції

}

В наведеній нижче програмі демонструється використання функцій. Параметри у функцію передаються за значенням (по-суті, така функція отримує копії значень тих *фактичних параметрів*, які стоять у виклику функції, і функція не може впливати на *фактичні параметри*, тобто не може їх змінювати)

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

// прототип
double f1( double, double );

int main()
{
    double a = 1.5;
    double b = 200.0;
    double result;

    setlocale(LC_CTYPE, "Ukrainian");
    printf("\nЗначення змінних a та b ПЕРЕД викликом ф-ції f1:\n");
    printf(" a=% .1lf\n", a);
    printf(" b=% .1lf\n", b);

    printf("\nВиклик ф-ції f1.");
    result = f1( a, b ); // виклик функції. a та b – фактичні параметри
                        // результат, який повертає функція в точку виклику
                        // присвоюється змінній result

    printf("\nРезультат=% .1lf\n\n", result);

    printf("\nЗначення змінних a та b ПІСЛЯ виклику ф-ції f1:\n");
    printf(" a=% .1lf\n", a);
    printf(" b=% .1lf\n", b);

    return 0;
}
```

```

//-----

// опис функції f1()
double f1( double a, double b) // перелік формальних параметрів
{
    a = 20.0;
    b = 50.0;

    return a+b;
}

```

На рис. 5.22 показані результати виконання приведеної програми.

```

Значення змінних a та b ПЕРЕД викликом ф-ції f1:
a=1.5
b=200.0

Виклик ф-ції f1.
Результат=20.0

Значення змінних a та b ПІСЛЯ виклику ф-ції f1:
a=1.5
b=200.0

```

Рис. 5.22 — Результат виконання програми

В наведеній вище програмі використовується функція `f1()`. Її прототип має такий вигляд:

```
double f1( double, double );
```

Таким чином, ця функція повинна отримувати в якості параметрів два значення типу `double` та повернати результат типу `double`.

В тілі функції `main()` ініціалізуються дві змінні типу `double` та оголошується змінна `result` типу `double`:

```
double a = 1.5;
double b = 200.0;
double result;
```

Початкове значення змінній `result` не задано, тому початкове значення цієї змінної невизначено. Змінній `result` буде присвоюватися значення, що буде повертатися функцією `f1()`.

Перед викликом функції `f1()` значення змінних `a` та `b` виводяться на екран в консольне вікно:

```
printf("\nЗначення змінних a та b ПЕРЕД викликом ф-ції f1:\n");
printf(" a=%lf\n", a);
printf(" b=%lf\n", b);
```

На рис. 5.22 видно, що `a=1.5`, `b=200.0`.

Після цього викликається функція `f1()`, якій передаються значення параметрів `a` та `b`, результат що повертає функція в точку виклику присвоюється змінній `result`:

```
result = f1( a, b );
```

Відбувається виконання функції `f1()`. Функція `f1()` має такий опис:

```
double f1( double a, double b )
{
    a = 20.0;
    b = 50.0;

    return a+b;
}
```

В заголовку функції записано перелік формальних параметрів:

```
double f1( double a, double b )
```

Параметри `a` та `b` типу `double` — це локальні змінні, які створюються при виклику функції `f1()`. Тобто, незважаючи на те, що імена цих параметрів співпадають із іменами параметрів `a` та `b`, які оголошенні у функції `main()`, але це різні змінні. Параметри з іменами `a` та `b`, що записані в заголовку функції `f1()` будуть зберігатися в пам'яті комп'ютера на період роботи функції `f1()`, по завершенню функції, ці параметри знищаться. Фактично, це можна описати такими кроками:

1. Викликається функція `f1()`.
2. Створюються локальні змінні `a` та `b` типу `double`, які є локальними змінними функції `f1()`, тобто видимі лише в цій функції, і знищуються по завершенню роботи функції `f1()`.

3. Параметрам *a* та *b* присвоюються значення фактичних параметрів, які стоять у виклику функції *f1()* (оскільки функція *f1()* викликається в тілі функції *main()* і функція *f1()* отримає в якості фактичних параметрів — значення змінних *a* та *b*, які оголошенні в тілі функції *main()*), то виходить так, що локальні змінні *a* та *b* функції *f1()*, — отримують значення параметрів *a* та *b*, що оголошенні в тілі *main()*). Таким чином, змінні *a* та *b*, що оголошенні в тілі функції *main()*, та змінні *a* та *b*, що виступають формальними параметрами і записані в заголовку функції *f1()* при її описі — це по суті різні змінні).

Таким чином, локальним змінним *a* та *b* функції *f1()* присвоюються значення фактичних параметрів, тому ці локальні змінні набувають таких значень: *a=1.5, b=200.0*.

4. В тілі функції *f1()* локальним змінним *a* та *b* присвоюються нові значення: *a=20.0, b=50.0*.

5. Оператор *return* повертає значення, що обчислюється у виразі *a+b*:

```
return a+b;
```

Отже, за допомогою оператора *return* в точку виклику повертається значення *70.0*.

Функція *f1()* завершила своє виконання. Локальні змінні *a* та *b* — знишились.

Після цього, у функції *main()* виконується виведення в консольне вікно значення змінної *result*, а також значення змінних *a* та *b*. Аналізуючи результати на рис. 5.22 видно, що значення змінних *a* та *b* залишились такими, якими вони були задані на початку програми. Таким чином, функція *f1()* не мала жодного впливу на ці змінні, а лише використовувала їх значення, тобто, отримувала копії значень змінних *a* та *b* і виконувала подальшу обробку відповідно до свого опису:

```
printf ("\nРезультат=%.\nlf\n", result);
```

```

printf("\nЗначення змінних а та b ПІСЛЯ виклику ф-ції f1:\n");
printf(" a=% .1lf\n", a);
printf(" b=% .1lf\n", b);

```

Загалом, ім'я формальних параметрів в заголовку функції `f1()` при опису функції можуть обиратися довільними, головне, щоб вони узгоджувалися із правилами складання ідентифікаторів. Не обов'язково надавати формальним параметрам ті ж самі імена, що імена фактичних параметрів у виклику функції.

Інший варіант передачі параметрів у функцію — це передача адрес параметрів, — тобто, використовуються вказівники. Функція отримує адреси, наприклад, відповідних змінних, і може виконувати доступ до значень, що зберігаються за цими адресами, а може і змінювати значення, які зберігаються за переданими адресами (якщо це не обмежується використанням службового слова `const`).

В наведеній нижче програмі демонструється використання функції, в якій параметри у функцію передаються за посиланням.

Результати виконання програми відображені на рис. 5.23, в консольному вікні відображені значення змінних `a` та `b` до та після виклику функції `x1()`.

```

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

// прототип
void x1( double *, double *, double * );

int main()
{
    double a = 1.5;
    double b = 200.0;
    double result;

    setlocale(LC_CTYPE, "Ukrainian");

    printf("\nЗначення змінних a та b ПЕРЕД викликом ф-ції x1:\n");
    printf(" a=% .1lf\n", a);
    printf(" b=% .1lf\n", b);

```

```

printf ("\nВиклик ф-ції x1.");

x1 (&a, &b, &result); // Виклик функції. У функцію передаються
// адреси змінних a, b та result

printf ("\nРезультат=% .1lf\n\n", result);

printf ("\nЗначення змінних a та b ПІСЛЯ виклику ф-ції x1:\n");
printf (" a=% .1lf\n", a);
printf (" b=% .1lf\n", b);

return 0;
}

//-----
// опис функції x1()
void x1( double *pa, double *pb, double *pr )
{
    *pa = 20.0;
    *pb = 50.0;

    *pr = *pa + *pb;
}

```

```

Значення змінних a та b ПЕРЕД викликом ф-ції x1:
a=1.5
b=200.0

Виклик ф-ції x1.
Результат=70.0

Значення змінних a та b ПІСЛЯ виклику ф-ції x1:
a=20.0
b=50.0

```

Рис. 5.23 — Результат виконання програми

В наведеній вище програмі прототип функції `x1()` має такий вигляд:

```
void x1( double *, double *, double * );
```

Функція `x1()` має тип `void`, отже це функція, що не повертає результат. В якості параметрів функція повинна отримувати 3 вказівники на `double`, —

передбачається, що функція в даній програмі буде отримувати адреси змінних типу double.

В тілі функції main() ініціалізуються дві змінні типу double, та оголошується змінна result типу double:

```
double a = 1.5;
double b = 200.0;
double result;
```

Початкове значення змінній result не задано, тому початкове значення цієї змінної невизначено. Змінній result буде надаватися деяке значення в процесі виконання функції x1().

Перед викликом функції x1() значення змінних a та b виводяться на екран в консольне вікно:

```
printf("\nЗначення змінних a та b ПЕРЕД викликом ф-ції x1:\n");
printf(" a=% .1lf\n", a);
printf(" b=% .1lf\n", b);
```

На рис. 5.23 видно, що a=1.5, b=200.0.

Після цього викликається функція x1(), якій передаються адреси змінних a, b та result. Оскільки функція x1() — це функція типу void, то на відміну від попереднього прикладу, ця функція не повертає ніякого результату:

```
x1( &a, &b, &result );
```

Відбувається виконання функції x1().

Функція x1() має такий опис:

```
void x1( double *pa, double *pb, double *pr )
{
    *pa = 20.0;
    *pb = 50.0;

    *pr = *pa + *pb;
}
```

В заголовку функції записано перелік формальних параметрів:

```
void x1( double *pa, double *pb, double *pr )
```

`ra`, `pb` та `pr` — це вказівники на `double`. Вказівник `ra` отримує адресу змінної `a`. Вказівник `pb` отримує адресу змінної `b`. Вказівник `pr` отримує адресу змінної `result`. Таким чином, вказівники `ra`, `pb` та `pr` зберігають адреси змінних `a`, `b` та `result`, які оголошенні в тілі функції `main()`. Тепер, маючи адреси вказаних змінних, можна виконувати не тільки доступ до значень цих змінних, але і змінювати ці значення, тобто надавати змінним `a`, `b` та `result` інших значень, ніж ті, які вони зберігали до виклику функції `x1()`. Це реалізується за допомогою використання вказівників.

Розіменування вказівника `ra` дозволяє отримати доступ до значення змінної `a`, і це значення змінної `a` перезаписується новим значенням, що дорівнює 20.0:

```
*ra = 20.0;
```

Розіменування вказівника `pb` дозволяє отримати доступ до значення змінної `b`, і це значення змінної `b` перезаписується новим значенням, що дорівнює 50.0:

```
*pb = 50.0;
```

Розіменування вказівника `pr` дозволяє отримати доступ до значення змінної `result`. До цього моменту значення змінної `result` було невизначенім. Тепер, в тілі функції `x1()` за допомогою вказівника `pr` змінна `result` буде визначатися як сума значень змінних `a` та `b`, причому доступ до значень змінних (значення яких попередньо були змінені на 20.0 та 50.0) забезпечується за допомогою вказівників `ra` та `pb`:

```
*pr = *ra + *pb;
```

Це останній рядок функції `x1()`, виконавши який, функція завершиться, вказівники `ra`, `pb` та `pr` — будуть знищені.

Після цього, у функції `main()` виконується виведення в консольне вікно значення змінної `result`, а також значення змінних `a` та `b`. Аналізуючи результати на рис. 5.23 видно, що значення змінних `a` та `b` змінилися, тобто

після виклику функції `x1()`, змінні `a` та `b` дорівнюють 20.0 та 50.0. Також відповідного значення набула змінна `result`, — якщо перед викликом функції `x1()` ця змінна була невизначена, то після виклику функції `x1()` змінна `result` дорівнює 70.0:

```
printf("\nРезультат=%.\n", result);  
  
printf("\nЗначення змінних a та b ПІСЛЯ виклику ф-ції x1:\n");  
printf(" a=%.\n", a);  
printf(" b=%.\n", b);
```

Загалом, вказівники представляють собою потужний інструментарій при реалізації відповідних алгоритмів. Вказівники активно використовуються не тільки при роботі із змінними, але також і при роботі із одновимірними та багатовимірними статичними та динамічними масивами.

Нижче представлено код програми, в якій описуються функції, прототипи яких були розглянуті на початку розділу. В програмі приводяться приклади реалізації різних функцій — з параметрами та без параметрів; функції, що повертають результат та функції типу `void`. Код програми носить демонстраційний характер для наочного представлення прикладів роботи із функціями.

```
#include <stdio.h>  
#include <stdlib.h>  
#include <locale.h>  
  
#define PI 3.14159  
  
void printHeader(void);  
void showResult( float amount );  
int getNumber( void );  
double areaCircle( double radius );  
void getCoef(int * plength, int * pwidth, int * pheight );  
  
  
int main()  
{  
    double R; // радіус кола  
    double S; // площа круга
```

```

int L, W, H; // L - довжина, W - ширина, H - висота
              // прямокутного паралелепіпеда
int V;       // V - об'єм прямокутного паралелепіпеда

int N; // кількість елементів в сумі ряду
       // який має вид: 1.0, 2.0, ..., N

float sum; // зберігає суму рядку 1.0 + 2.0 + ... + N

int i;

setlocale(LC_CTYPE, "Ukrainian");

printf("\nВведіть значення N для обрахунку суми 1.0 + 2.0 +... +N ");
N = getNumber();

sum = 0.0;
for( i = 0; i <= N; i++ )
    sum += (float)i;

printf("\nВведіть значення радіуса для обрахунку площини круга \n");
do
{
    printf("Radius=");
    scanf("%lf", &R);
}while( R <= 0 );

S = areaCircle( R );

printf("\nВведіть Довжину(L) Ширину(W) та Висоту(H) ");
printf("\nдля обрахунку об'єму прямокутного паралелепіпеда");
printf("\nL, W, H - цілі додатні значення\n");
printf("\nПри введенні значень МЕНШЕ 1 – завершення програми\n");
getCoef( &L, &W, &H);

V = L * W * H;

// Виведення результатів
printHeader();

printf("\nСума ряду 1.0 + 2.0 +... +N:");
showResult( sum );

printf("\n\nПлоща круга = %.4lf\n", S);

printf("\nОб'єм прямокутного паралелепіпеда:");
printf("\nV=%d\n", V);
return 0;
}

```

```

//----- Опис функцій -----

```

```

void printHeader(void)
{
    printf("\n\n-----\n");
    printf("\t\t\tРЕЗУЛЬТАТИ:\n\n");
}

//-----

void showResult( float amount )
{
    printf("\nСума: %.2f\n", amount);
}

//-----

int getNumber( void )
{
    int temp;

    printf("\n\nВведіть ціле значення більше 0: \n");
    scanf("%d", &temp);

    while( temp <= 0 ){
        printf("Введіть ціле значення ВІЛЪШЕ 0: \n");
        scanf("%d", &temp);
    }

    return temp;
}

//-----

double areaCircle( double radius )
{
    return PI * radius * radius;
}

```

```

void getCoef( int * plength, int * pwidth, int * pheight )
{
    printf("\nL=");
    scanf("%d", plength);

    if( *plength <= 0 ){
        printf("\nНедопустиме значення!!!!");
        printf("\nНатисніть клавішу для завершення програми");
        getch();
        exit(EXIT_SUCCESS);
    } else {
        printf("\nW=");
        scanf("%d", pwidth);

        if( *pwidth <= 0 ){
            printf("\nНедопустиме значення!!!!");
            printf("\nНатисніть клавішу для завершення програми");
            getch();
            exit(EXIT_SUCCESS);
        } else {
            printf("\nH=");
            scanf("%d", pheight);
        }
    }
}

```

5.4 Питання для самоконтролю

1. Що таке вказівник?
2. Яка відмінність між передачею у функцію параметрів за значенням та за посиланням?
3. Навіщо використовується оператор `return` у функції типу `void`?
4. Значення якого типу буде повертати функція, що має прототип `int f(double)`?
5. Чи можна написати функцію, яка не отримує жодних параметрів, але повертає результат в точку виклику?
6. Чи можна вказівнику на `void` присвоїти адресу змінної цілого типу даних?

6. ОДНОВИМІРНІ ТА БАГАТОВИМІРНІ СТАТИЧНІ МАСИВИ

Масиви використовуються для зберігання однотипних даних.

Масив — це сукупність даних одного типу, які послідовно розміщуються в пам'яті, і мають спільне ім'я [5].

Масиви можуть використовуватися, наприклад, для збереження деяких вимірів, показників певних датчиків, відліків сигналу тощо. Використання масивів може знадобитися при виконанні різноманітних розрахунково-обчислювальних процедур, які необхідно застосовувати при реалізації відповідних алгоритмів, наприклад, в задачах статистичного аналізу даних, фільтрації даних тощо.

По характеру представлення даних масиви можна розділити на [5, 6]:

- одновимірні масиви;
- багатовимірні масиви.

6.1 Одновимірний статичний масив. Принципи роботи із одновимірними статичними масивами

Одновимірний масив може бути представлений у вигляді вектора-рядка, де кожен елемент масиву має свій індекс. Індексація елементів масиву в мові С починається з нуля. Таким чином, схематично одновимірний масив, який, наприклад, складається із шести елементів, де кожен елемент зберігає дійсне число, можна представити відповідно до схеми, що показана на рис. 6.1 [4, 7].

[0]	[1]	[2]	[3]	[4]	[5]
1.75	-2.05	-0.12	0.58	1.05	2.10

Рис. 6.1 — Приклад одновимірного масиву із шести елементів

Схематично зображеній на рис. 6.1 одновимірний масив має розмір 6, тобто складається із шести елементів, а індекси відповідних елементів масиву знаходяться в проміжку 0...5. Таким чином, щоб звернутися до потрібного елементу масиву потрібно вказати ім'я масиву та індекс відповідного елементу.

Ім'я масиву — це ідентифікатор, тому при виборі імені потрібно керуватися правилом складання ідентифікаторів. Наприклад, якщо задано масив, що має ім'я `testArray` для якого обрано відповідний тип даних, наприклад, `float`, то щоб звернутися до *першого* елементу масиву необхідно звернутися до елемента з індексом 0: `testArray[0]`; щоб звернутися до *другого* елементу масиву необхідно вказати індекс 1: `testArray[1]`; щоб звернутися до *третього* елементу масиву необхідно вказати індекс 2: `testArray[2]` і т.д.

Враховуючи, що масив представляє собою набір із однотипних даних, то для обробки елементів масиву часто потрібно виконувати певний набір дій, який буде однаковим при обробці кожного окремого елементу. Таким чином, щоб виконати обробку всіх елементів масиву, часто таку обробку необхідно виконувати із застосуванням операторів циклу. В тілі циклу записується набір операторів, що реалізують відповідний алгоритм обробки, який застосовується до кожного окремого елементу масиву. Шляхом зміни індексу елементу масиву відбувається перехід до наступного елемента масиву. Змінюючи (збільшуєчи або зменшуючи) значення індексу елементу на кожній ітерації циклу, можна виконати обробку всіх елементів масиву.

При оголошенні масиву потрібно вказати тип даних, імя масиву та кількість елементів в масиві [3—8]. Наприклад:

```
float testArray[6];
```

Масив має ім'я `testArray`, призначений для зберігання значень типу `float`, розмір масиву (або іншими словами — кількість елементів в масиві) становить 6. В даному випадку, масив лише оголошується, але елементи масиву не ініціалізуються, тобто не вказується значення елементів масиву, тому, по аналогії із локальними змінними, які при оголошенні мають невизначене

значення, так і елементи масиву є невизначені, якщо масив оголошується в тілі деякої функції, наприклад, в тілі функції `main()`.

Для того щоб виконати ініціалізацію елементів масиву, необхідно записати у фігурних дужках значення елементів масив, відділяючи їх комою. Наприклад, в наступному рядку відбувається ініціалізація всіх елементів масиву:

```
float testArray[6] = {1.75, -2.05, -0.12, 0.58, 1.05, 2.10};
```

У випадку, якщо потрібно задати нульове значення всім елементам масиву, то достатньо у фігурних дужках вказати 0:

```
float testArray[6] = { 0 };
```

Якщо при ініціалізації елементу масиву у фігурних дужках вказати якебудь відмінне від нуля значення, то лише перший елемент масиву буде дорівнювати цьому значенню, а всі інші елементи масиву — будуть дорівнювати нулю:

```
float testArray[6] = { -9.5 };
```

Перший елемент масиву `testArray[0]` = -9.5 , а всі інші елементи — будуть дорівнювати нулю: `testArray[1]=0.0; testArray[2]=0.0;... testArray[5]=0.0`.

Якщо при ініціалізації елементів масиву в фігурних дужках буде записано менша кількість значень, ніж розмір масиву, то всім елементам, для яких в явному вигляді не були записані відповідні значення, буде присвоєно 0. Наприклад:

```
float testArray[6] = {1.75, -2.05, -0.12};
```

В такому випадку: `testArray[0]` = -1.75 , `testArray[1]` = -2.05 , `testArray[2]` = -0.12 , `testArray[3]` = 0.0 , `testArray[4]` = 0.0 , `testArray[5]` = 0.0 .

При ініціалізації масиву розмір масиву може опускатися, в такому випадку, необхідна кількість пам'яті, яка потрібна для зберігання масиву буде визначатися на основі кількості записаних значень в фігурних дужках:

```
float array2[ ] = {1.75, -2.05, -0.12 };
```

Масив `array2` складається із трьох елементів. Для визначення в програмі розміру такого масиву, для кого в явному вигляді розмір не заданий, можна скористатися операцією `sizeof` [5, 7].

Якщо потрібно, щоб значення елементів масиву, які були ініціалізовані, не можна було змінювати на етапі виконання програми, потрібно перед типом даних записати службове слово `const`, що дозволить розглядати такий масив як масив констант:

```
const float array2[ 3 ] = {1.75, -2.05, -0.12 };
```

Одновимірні масиви, які наведені в прикладах, — це одновимірні *статичні* масиви. Статичні масиви передбачають визначення розміру масиву на етапі написання програми. Розмір в статичних масивах повинен бути константою. За допомогою директиви препроцесора `#define` можна створити відповідну константу, яка буде призначена для зберігання розміру масиву. Розмір масиву — це ціле додатне число.

При роботі з масивами важливо пам'ятати, що виходити за межі масиву заборонено, тобто не можна звертатися до неіснуючих елементів масиву, шляхом встановлення значення індексу елементу більшим ніж індекс останнього елементу масиву. Контроль виходу за межі масиву покладається виключно на розробника програми. Тому важливо приділяти прискіпливу увагу при формування виразів-умов в операторах циклу, які використовуються при роботі з масивами і передбачають врахування значення індексу.

Нижче показано приклад програми в якій визначається сума значень елементів масиву. При ініціалізації масиву розмір масиву в явному вигляді не вказаний. В програмі визначається розмір масиву за допомогою операції `sizeof`. Застосовуючи цю операцію до імені масиву можна визначити сукупну кількість байт пам'яті, яка була виділена для статичного масиву. Застосовуючи операцію `sizeof` до першого елемента масиву (це елемент масиву з індексом 0), можна визначити кількість байт пам'яті, яка виділена для збереження одного значення

в масиві. Поділивши перше значення на друге, можна дізнатися кількість елементів в масиві. Зберігши це значення в певній змінній, можна її використовувати у виразі-умові в операторі циклу, щоб контролювати поточне значення індексу елементу масиву, щоб не виходити за межі масиву:

```
#include <conio.h>
#include <stdio.h>
#include <locale.h>

int main(void)
{
    float testArray[ ]={1.75, -2.05, -0.12, 0.58}; // масив
    float sum = 0.0; // змінна призначена для зберігання суми
                    // значень елементів масиву testArray

    unsigned int oneElementBytes; // кількість байтів пам'яті, яка виділена
                                // для зберігання одного елементу масиву

    unsigned int arrayBytes; // кількість байтів пам'яті, яка виділена
                            // для зберігання всього масиву

    unsigned int size; // змінна для зберігання кількості елементів масиву
    unsigned int index; // змінна для зберігання поточного індексу елементу

    oneElementBytes = sizeof testArray[0]; // к-сть байт для одного елементу

    arrayBytes = sizeof testArray; // к-сть байт для всього масиву

    size = arrayBytes / oneElementBytes; // к-сть елементів в масиві

    setlocale(LC_CTYPE, "Ukrainian");
    printf("\nК-ість Байт для одного елементу = %u", oneElementBytes);
    printf("\nК-ість Байт для всього масиву = %u", arrayBytes);
    printf("\nРозмір масиву = %u", size);
    printf("\n\n");
    printf("Значення елементів масиву:\n");

    for( index = 0; index < size; index++ ){
        printf("testArray[ %u ] =% .2f \n", index, testArray[index] );
        sum += testArray[index];
    }

    printf("\n");
    printf("Сума елементів масиву:\n%.2f", sum);
    return 0;
}
```

На рис. 6.2 показано результат виконання програми.

```
Кількість байт для одного елементу = 4
Кількість байт для всього масиву = 16
Розмір масиву = 4

Значення елементів масиву:
testArray[ 0 ] = 1.75
testArray[ 1 ] = -2.05
testArray[ 2 ] = -0.12
testArray[ 3 ] = 0.58

Сума елементів масиву:
0.16
```

Рис. 6.2 — Знаходження суми елементів одновимірного статичного масиву

Статичні масиви можна передавати у функцію в якості параметра. Потрібно запам'ятати, що ім'я статичного масиву — це вказівник на перший елемент масиву. Причому, це константний вказівник, який завжди вказує на перший елемент масиву [3, 6]. При передачі масиву в якості параметра функції необхідно записати ім'я масиву, а також можна в якості другого параметра передавати розмір масиву. Враховуючи, що ім'я масиву — це вказівник на перший елемент, то значення елементів масиву можна змінювати в тілі відповідної функції. Якщо потрібно забезпечити неможливість зміни значень елементів масиву в тілі відповідної функції, яка виконує обробку масиву, то в такому випадку в прототипі такої функції, а також в заголовку такої функції при її описі необхідно використати службове слово `const`. Це призведе до того, що відповідна функція, яка отримує масив в якості параметра, буде сприймати цей масив як масив констант, а константи змінювати на етапі виконання програми заборонено. Тому бажано тим функціям, які по логіці своєї роботи не повинні змінювати значення елементів масиву, забороняти це робити шляхом застосування службового слова `const`. Прототип функції та заголовок функції, яка отримує в якості параметра одновимірний статичний масив, записується аналогічним чином як це робилося для випадку передачі у функцію значень змінних чи констант, за виключенням того, що необхідно крім типу даних масиву також записати квадратні дужки. В прикладі, який представлений нижче, виконується обробка одновимірного статичного масиву типу `double`.

Функція `print_1D_array()` призначена для виведення значень елементів масиву в консольне вікно. Функція `get_1D_array()` призначена для введення значень елементів масиву з клавіатури. Функція `find_max_in_1D_array()` призначена для знаходження максимального значення в одновимірному статичному масиві. Кожна із функцій буде отримувати в якості параметрів ім'я масиву та розмір масиву. Враховуючи, що функції `print_1D_array()` та `find_max_in_1D_array()` не повинні в ході своєї роботи виконувати зміну значень елементів масиву, то в їхніх прототипах та заголовках записується службове слово `const`, тим самим роблячи неможливим зміну значень елементів масиву в тілі цих функцій:

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

#define N 4

void get_1D_array( double [], int );
void print_1D_array( const double [], int );
double find_max_in_1D_array( const double [], int );

int main()
{
    double A[N] ={0.0}; // Одновимірний масив типу double
                        // Всім елементам присвоюється значення 0.0

    double maxValueInArray; // змінна призначена для збереження
                           // знайденого максимального значення в масиві

    get_1D_array( A, N ); // виклик функції для присвоєння елементам
                          // масиву A значень, які вводяться з клавіатури

    print_1D_array( A, N ); // виклик функції для виведення значень
                           // елементів масиву на екран

    maxValueInArray = find_max_in_1D_array( A, N ); // знаходження
                                                    // максимального
                                                    // значення в масиві

    setlocale(LC_CTYPE, "Ukrainian");
    printf("\n\nМаксимальне значення в масиві:\n");
    printf("%.3lf \n\n", maxValueInArray);

    return 0;
}
```

```

//-----

void get_1D_array( double a[], int n )
{
    int j; // змінна яка буде використовуватися для зберігання
           // поточного індекса елемента масиву при введенні значень
           // елементів масиву з клавіатури

    printf("\nМасив із %d елементів типу double", n);
    printf("\nВведіть значення елементів масиву:\n");

    for( j = 0; j <= n-1; j++ ){
        printf("A[ %d ] = ", j);
        scanf("%lf", &a[j] );           // введення значення елементу масиву
    }
}

//-----
```

```

void print_1D_array( const double a[], int n )
{
    int j;

    printf("\n\nМасив:");

    for( j = 0; j < n; j++ ){
        printf("\nA[%d]=%.3lf", j, a[j] );
    }
}
```

```

//-----
```

```

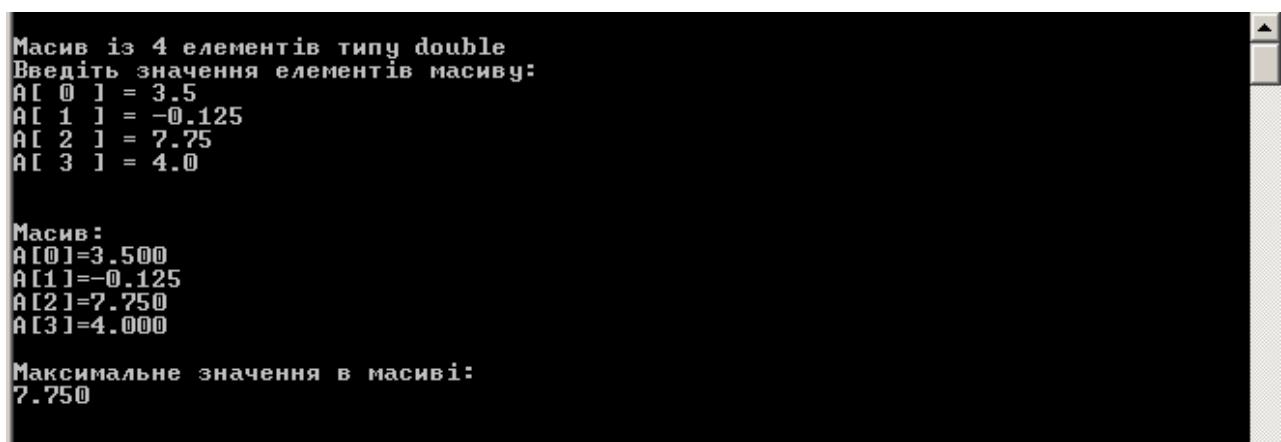
double find_max_in_1D_array( const double a[], int n )
{
    int j;
    double tempMax;

    j=0;
    tempMax = a[j];

    for( j = 1; j < n; j++ )
        if( a[j] > tempMax )
            tempMax = a[j];

    return tempMax;
}
```

В функціях `get_1D_array()`, `print_1D_array()` та `find_max_in_1D_array()`, в яких виконувалася відповідна обробка масиву, розмір масиву передавався як окремий параметр, що присвоювався локальній змінній `n`. Значення цієї змінної використовувалося в операторі циклу `for`, в якому на кожній ітерації відбувалася зміна значення індексу поточного елементу масиву (значення індексу елементу масиву зберігалося в змінній на ім'я `j`). Враховуючи, що `n` — це розмір масиву, а індекс останнього елементу масиву на одиницю менший ніж розмір масиву, тому в операторі циклу `for` використовувався відповідний вираз-умова з метою контролю значення індексу елементу, щоб цикл виконувався доти, поки індекс елементу знаходиться в допустимих межах, тобто щоб не відбувався вихід за межі масиву. В одній функції в циклі `for` застосовувалася умова `j < n;` (тобто максимальне значення індексу `j` при якому дана умова залишалася істинною — це `j = n-1`). В іншій функції в циклі `for` застосовувалася умова `j <= n-1;` (яка по-суті, аналогічна попередній умові, в якій застосовувалася операція «Менше», оскільки максимальне значення індексу `j` при якому дана умова залишалася істинною — це також `j = n-1`). Результати виконання приведеної вище програми показані на рис. 6.3.



```
Масив із 4 елементів типу double
Введіть значення елементів масиву:
A[ 0 ] = 3.5
A[ 1 ] = -0.125
A[ 2 ] = 7.75
A[ 3 ] = 4.0

Масив:
A[0]=3.500
A[1]=-0.125
A[2]=7.750
A[3]=4.000

Максимальне значення в масиві:
7.750
```

Рис. 6.3 — Результат виконання програми по знаходженню максимального значення в одновимірному статичному масиві

6.2 Багатовимірний статичний масив. Принципи роботи із багатовимірними статичними масивами на прикладі двовимірного масиву

Крім одновимірних статичних масивів, широке застосування знаходить багатовимірні статичні масиви. Якщо одновимірний масив можна представити у вигляді вектора-рядка, то двовимірний масив (який є окремим випадком багатовимірного масиву) можна (для зручності) представити у вигляді матриці (квадратної чи прямокутної) (рис. 6.4) [3—5].

Індекси стовпців						
	[0]	[1]	[2]	[3]	[4]	
Індекси рядків	[0]	$[0][0] = 5.25$	$[0][1] = -0.97$	$[0][2] = 3.05$	$[0][3] = 2.74$	$[0][4] = 7.21$
	[1]	$[1][0] = -1.18$	$[1][1] = 6.02$	$[1][2] = 0.94$	$[1][3] = -8.02$	$[1][4] = 4.65$
	[2]	$[2][0] = 2.37$	$[2][1] = -9.04$	$[2][2] = -1.55$	$[2][3] = 7.08$	$[2][4] = 3.98$
	[3]	$[3][0] = 5.82$	$[3][1] = -0.95$	$[3][2] = 4.81$	$[3][3] = -2.43$	$[3][4] = 1.84$
	[4]	$[4][0] = -0.45$	$[4][1] = 2.34$	$[4][2] = 5.19$	$[4][3] = -4.71$	$[4][4] = 9.15$

Рис. 6.4 — Представлення двовимірного статичного масиву у вигляді матриці

Кожен елемент двовимірного масиву має два індекси — індекс рядка та індекс стовпця (або іншими словами індекс позиції елемента у відповідному рядку). Тип даних значень, які зберігають елементи масиву визначається при оголошенні масиву. Двовимірний масив, який схематично показаний у вигляді матриці, в пам'яті комп'ютера зберігається у вигляді довгого вектора-рядка, який сформований із рядків показаної на рис. 6.4 матриці, тобто в пам'яті комп'ютера спочатку розміщуються елементи першого рядка масиву, потім елементи другого рядка, після цього елементи третього рядка і т.д. Схематичний варіант зберігання двовимірного масиву в пам'яті комп'ютера показано на рис. 6.5 [7].

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	...	[4][0]	[4][1]	[4][2]	[4][3]	[4][4]
5.25	-0.97	3.05	2.74	7.21	-1.18	6.02	0.94	-8.02	4.65	...	-0.45	2.34	5.19	-4.71	9.15

Рядок з індексом 0
(перший рядок матриці)

Рядок, з індексом 1
(другий рядок матриці)

Рядок, з індексом 4
(п'ятий рядок матриці)

Рис. 6.5 — Розміщення елементів статичного двовимірного масиву в пам'яті комп'ютера

При оголошенні двовимірного масиву розміри відповідних вимірів повинні задаватися у вигляді константи, при цьому можна скористатися директивною препроцесора `#define` для створення відповідних констант [5].

Аналогічно одновимірним масивам, елементам двовимірного масиву можуть бути задані значення на етапі оголошення масиву (ініціалізація елементів масиву).

Нехай в контексті інтерпретації двовимірного статичного масиву у вигляді матриці, кількість рядків в масиві визначається константою N , а кількість стовпців — визначається константою M , де значення для N та M задаються, наприклад, наступним чином:

```
#define N 3  
#define M 5
```

Розглянемо варіанти записів при оголошенні та ініціалізації двовимірного масиву.

При оголошенні масиву вказується тип даних масиву, ім'я масиву та значення для відповідного виміру, — кількість рядків та стовпців [3—8]:

```
int b[N][M];
```

Таким чином, в програмі створюється масив на ім'я *b*, що складається із *N* рядків та *M* стовпців (або можна виконати інтерпретацію по-іншому — масив складається із *N* рядків, де в кожному рядку розміщується *M* елементів). При такому записі значення елементів масиву не ініціалізуються, тому значення елементам масиву можна задати на етапі виконання програми, ввівши їх з клавіатури, або присвоївши елементам масиву значення обрахунку деякого виразу тощо.

Щоб звернутися до деякого елементу масиву, наприклад, з метою задати значення відповідному елементу, необхідно вказати ім'я масиву та індекси елементу. Наприклад, елемент *b[0][0]* — це перший елемент первого рядка, елемент *b[2][4]* — це п'ятий елемент третього рядка. Потрібно пам'ятати, що індексація елементів масиву починається з 0.

Для обробки масивів найчастіше доводиться використовувати оператори циклу, в яких, змінюючи значення відповідних індексів, можна виконувати обробку всіх елементів масиву. Потрібно завжди пам'ятати про необхідність контролювати вихід за межі масиву.

В наступному рядку показаний варіант запису, при якому виконується ініціалізація всіх елементів масиву, який складається із 3-х рядків, де в кожному рядку 5 елементів (в контексті представлення двовимірного масиву у вигляді матриці, то даний масив буде представляти собою матрицю 3×5):

```

int b[N][M] = {
    { 5, 2, 7, 4, 1 },
    { 9, 3, 6, 8, 3 },
    { 1, 7, 2, 4, 9 }
};

```

Елементи кожного рядка записані в фігурних дужках, причому самі рядки також записані між відкриваючою та закриваючою фігурними дужками. Нижче представлено розміщення елементів при в контексті розгляду масиву як матриці розміром 3×5 :

5	2	7	4	1
9	3	6	8	3
1	7	2	4	9

Якщо при ініціалізації елементів масиву будуть в явному вигляді записана кількість значень, яка менша ніж кількість елементів, то відповідні елементи, для яких не вказано числові значення, будуть отримувати значення 0:

```

int b[N][M] = {
    { 5, 2 },
    { 9, 3, 6 },
    { 1 }
};

```

Значення відповідних елементів масиву в даному випадку наступні:

5	2	0	0	0
9	3	6	0	0
1	0	0	0	0

При ініціалізації елементів двовимірного масиву можна не виконувати запис значень, групуючи їх в фігурних дужках для кожного рядку, а записувати їх аналогічно як це робилося для одновимірного масиву, при цьому заповнення значень елементів буде відбуватися послідовно починаючи з першого елемента першого рядка. Наприклад:

```
int b[N][M] = { 5, 2, 7, 4, 1, 9, 3, 6, 8, 3, 1, 7, 2, 4, 9 };
```

Значення відповідних елементів масиву для наведеного прикладу:

5	2	7	4	1
9	3	6	8	3
1	7	2	4	9

Якщо при попередньому варіанті ініціалізації масиву кількість значень в переліку у фігурних дужках буде меншою, ніж загальна кількість елементів, то відповідним елементам масиву, для яких не було вказано в явному вигляді числове значення, буде присвоєно 0. Наприклад:

```
int b[N][M] = { 5, 2, 7, 4, 1, 9, 3, 6, 8, 3, 1 };
```

Значення відповідних елементів масиву для наведеного прикладу:

5	2	7	4	1
9	3	6	8	3
1	0	0	0	0

Таким чином, для ініціалізації нульовими значенням всіх елементів масиву достатньо записати так:

```
int b[N][M] = { 0 };
```

В цьому випадку всі елементи масиву будуть дорівнювати 0:

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

В наступному прикладі приведена програма, яка виводить значення елементів двовимірного масиву у вигляді матриці. Для реалізації цього завдання необхідно скористатися оператором циклу `for`, причому використовуються два оператори циклу — один оператор `for` вкладений в інший. В зовнішньому циклі виконується зміна значень індексу рядка, а у вкладеному циклі `for` відбувається зміна значення індексу стовпця (або іншими словами — відбувається зміна значення індексу елемента в межах поточного рядка). В наведеному прикладі змінна `i` використовується як індекс рядка,

змінна *j* використовується як індекс стовпця. Після виведення значень рядка, що відбувається у вкладеному циклі *for*, виконується вихід із вкладеного циклу *for*, наступний оператор, який виконується — це виклик функції *printf("\n")*, — ця функція використовується для переведення курсору на новий рядок на екрані в консольному вікні, після цього відбувається збільшення на 1 значення індексу рядка, виконується перевірка умови *i < N*, що призначена для контролю значення індексу поточного рядка, і якщо умова істинна, то виконується тіло циклу, яке складається із внутрішнього оператора *for* та виклику функції *printf("\n")*:

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

#define N 3
#define M 5

int main()
{
    int b[N][M] = {
        { 5, 2, 7, 4, 1 },
        { 9, 3, 6, 8, 3 },
        { 1, 7, 2, 4, 9 }
    };

    int i, j;

    for( i=0; i<N; i++ ) {
        for(j=0; j<M; j++)
            printf("%3d", b[i][j]);
        printf("\n");
    }

    return 0;
}
```

На рис. 6.6 представлено результат виведення значень елементів двовимірного масиву у вигляді матриці.

```
5 2 7 4 1
9 3 6 8 3
1 7 2 4 9
Process returned 0 <0x0> execution time : 0.010 s
Press any key to continue.
```

Рис. 6.6 — Виведення значень елементів двовимірного масиву у вигляді матриці

Багатовимірні масиви так само як і одновимірні можуть бути передані у функцію в якості параметру.

При записі прототипів функцій, які в якості параметру отримують багатовимірний масив, потрібно пам'ятати, що в квадратних дужках необхідно вказувати значення відповідних вимірів починаючи з другого. При написанні функцій по роботі з масивами можна використовувати службове слово `const` в записі прототипу та заголовку функції при її описі, щоб функція не мала права змінювати значення елементів масиву при його обробці.

Нижче приведений прототип функції, яка призначена для виведення значень двовимірного масиву типу `int` у вигляді матриці:

```
void print_2D_int_array( const int [ ] [M], int row, int col );
```

В дану функцію буде передаватися не тільки двовимірний масив типу `int`, але також і два параметра типу `int`, які відповідають за кількість рядків (параметр `row`) та кількість стовпців (параметр `col`). В прототипах функцій імена параметрів ігноруються, але в даному випадку імена були спеціально записані, щоб дати зрозуміти за що буде відповіти кожен із цілих значень, які будуть передаватися у функцію разом з масивом. Крім того, щоб випадково не записати невірне значення для відповідного виміру, при записі прототипу функції, яка отримує двовимірний (багатовимірний) масив, можна також записати значення в квадратних дужках і для першого виміру:

```
void print_2D_int_array( const int [N] [M], int row, int col );
```

Враховуючи, що N та M в наведеному прикладі програми, — це глобальні константи, тому доступ до цих значень, які відповідають кількості рядків та кількості стовпців в масиві, є всюди нижче по тексту програми, але щоб зробити функцію більш загальною, яку можна буде використати, наприклад, при роботі із динамічними двовимірними масивами типу `int` шляхом внесення у прототип та заголовок функції мінімальних змін, — значення кількості рядків та стовпців передаються у функцію у вигляді окремих параметрів.

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

#define N 3
#define M 5

void print_2D_int_array( const int [N][M], int row, int col );

int main()
{
    int b[N][M] = {
        { 5, 2, 7, 4, 1 },
        { 9, 3, 6, 8, 3 },
        { 1, 7, 2, 4, 9 }
    };

    print_2D_int_array( b, N, M );

    return 0;
}

//-----

void print_2D_int_array( const int b[N][M], int row, int col )
{
    int i, j;

    for( i=0; i<N; i++ ) {
        for( j=0; j<M; j++ )
            printf("%3d", b[i][j]);

        printf("\n");
    }
}
```

Результати виконання програми показані на рис. 6.6.

Наступний приклад демонструє роботу із двовимірними масивами. Покладається, що двовимірний масив на ім'я `inputData` використовується для збереження значень вимірювань деяких датчиків. Масив можна представити у вигляді матриці розміром `SENSOR × MEASURE`, де `SENSOR` та `MEASURE` — це константи. `SENSOR` — це кількість датчиків. `MEASURE` — кількість вимірювань, які зроблені кожним датчиком. Таким чином, по умові завдання — в першому рядку масиву `inputData` містяться виміри первого датчика, в другому рядку — містяться виміри другого датчика, в третьому рядку — містяться виміри третього датчика. Функція `initInputData()` використовується для введення значень вимірів датчиків з клавіатури. Функція `printInputData()` використовується для виведення введених показників в консольне вікно у вигляді таблиці. В функції `main()` створюються два масиви — `mSens` та `mMea`, — перший масив призначений для зберігання середнього значення вимірювань для кожного датчика, а другий масив призначений для зберігання середнього значення для кожного вимірювання на основі даних з усіх датчиків. Для цього використовуються функції `meanForSensor()` та `meanForMeasure()`. Для виведення в консольне вікно обрахованих значень для кожного із масивів використовуються відповідно функції `printMeanForSensor()` та `printMeanForMeasure()`. Нижче наводиться текст програми. В тексті програми надані деякі коментарі:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <locale.h>

#define SENSOR 3
#define MEASURE 6
// SENSOR — кількість датчиків для вимірювання деякого показника
// MEASURE — кількість вимірювань для кожного датчика

void initInputData( int [SENSOR][MEASURE], int, int );
void printInputData( const int [SENSOR][MEASURE], int, int );

void meanForSensor( float [SENSOR], const int [SENSOR][MEASURE], int, int );
void printMeanForSensor( const float [SENSOR], int );

void meanForMeasure( float [MEASURE], const int [SENSOR][MEASURE], int, int );
void printMeanForMeasure( const float [MEASURE], int );
```

```

int main()
{
    int inputData[SENSOR][MEASURE] = { 0 }; // Двовимірний масив, який
                                            // призначений для зберігання вимірювань.
                                            // В першому рядку зберігаються показники
                                            // 1го датчика. В 2му рядку зберігаються
                                            // показники 2го датчика. В 3му рядку
                                            // зберігаються показники 3го датчика.
                                            // Значення вимірів вводяться з клавіатури.
                                            // Початкові значення елементів масиву
                                            // дорівнюють 0.

    float mSens[SENSOR] = { 0.0 }; // Одновимірний масив, який
                                    // призначений для зберігання обрахованого
                                    // середнього значення вимірів для кожного
                                    // датчика. 1й елемент масиву зберігає
                                    // середній показник вимірів для 1го датчика.
                                    // 2й елемент масиву зберігає середній
                                    // показник вимірів для 2го датчика.
                                    // 3й елемент масиву зберігає середній
                                    // показник вимірів для 3го датчика.
                                    // Початкове значення елементам масиву
                                    // задається рівним 0.0.

    float mMea[MEASURE] = { 0.0 }; // Одновимірний масив, який
                                    // призначений для зберігання обра-
                                    // хованого середнього значення для
                                    // кожного вимірювання на основі показників
                                    // отриманих з усіх трьох датчиків.
                                    // 1й елемент масиву зберігає середнє значе-
                                    // для 1го вимірювання, отриманого з усіх
                                    // 3х датчиків. 2й елемент масиву зберігає
                                    // середнє значення 2го вимірювання, отрима-
                                    // ного з усіх 3х датчиків. І т.д.
                                    // Початкове значення елементам масиву
                                    // задається рівним 0.0.

    setlocale(LC_CTYPE, "Ukrainian");

    // Елементам масиву задаються вхідні значення
    initInputData( inputData, SENSOR, MEASURE );

    // Виведення значень масиву в консольне вікно
    printInputData( inputData, SENSOR, MEASURE );

    // Визначення середніх показників вимірів для
    // кожного датчика.
    meanForSensor( mSens, inputData, SENSOR, MEASURE );

    // Виведення значень середніх показників вимірів,
    // отриманих для кожного датчика, на екран в
    // консольне вікно.
    printMeanForSensor( mSens, SENSOR );
}

```

```

    // Визначення середніх показників для кожного виміру
    // на основі показників з трьох датчиків.
    meanForMeasure( mMea, inputData, SENSOR, MEASURE );

    // Виведення значень середніх показників вимірів,
    // отриманих для кожного виміру, на основі показників,
    // які отримані з усіх трьох датчиків, на екран в
    // консольне вікно.
    printMeanForMeasure( mMea, MEASURE );

    return 0;
}

```

//----- Опис функцій -----

```

void initInputData( int data[SENSOR][MEASURE], int sens, int numMea )
{

```

```

    int s; // індекс рядка ( номер датчика )
    int m; // індекс стовпця ( номер вимірювання )


```

```

    printf("\n\nДля кожного датчика введіть %d вимірів.", numMea );
    printf("\nВсього датчиків - %d шт. ", sens);
    printf("\nЗначення виміру датчика - це ціле число. \n");

```

```

    for( s = 0; s < sens; s++ ) {
        printf("\n---\nДатчик номер %d:\n", s);
        for( m = 0; m < numMea; m++ ) {
            printf("Вимір[%d] [ %d ] = ", s, m );
            scanf("%d", &data[s][m] );
        }
    }
}

```

// -----

```

void printInputData( const int data[SENSOR][MEASURE], int sens, int numMea )
{

```

```

    int s; // індекс рядка ( номер датчика )
    int m; // індекс стовпця ( номер вимірювання )


```

```

    printf("\n\n\n");

```

```

    printf("%12s", " ");
    for( m = 0; m < numMea; m++ )
        printf("%6s[%d]", "Вим.", m );

```

```

    for( s = 0; s < sens; s++ ) {
        printf("\nДатчик[%d]: ", s );
        for( m = 0; m < numMea; m++ ) {
            printf("%5c%-4d", ' ', data[s][m] );
        }
        printf("\n");
    }
}

```

//-----

```

void meanForSensor(float ms [SENSOR], const int data[SENSOR] [MEASURE], int S,
int M)
{
    int i; // індекс рядка ( номер датчика )
    int j; // індекс стовпця ( номер вимірювання )
    int sum; // сума вимірювань для обрахунку середнього значення
              // вимірювання для кожного датчика

    for( i = 0; i < S; i++) {
        sum = 0;
        for( j = 0; j < M; j++ )
            sum += data[i][j];

        ms[i] = (float)sum / (float)M;
    }

}

//-----

void printMeanForSensor( const float mSens [SENSOR], int sens )
{
    int i; // індекс елемента

    printf("\n\n\n");
    printf("-----\n");
    printf("Середні значення вимірювань для кожного датчика: ");

    for( i = 0; i < sens; i++ )
        printf("\nДатчик[%d]: %7.2f", i, mSens[i] );

    printf("\n");
}

//-----


void meanForMeasure(float mMea [MEASURE], const int data[SENSOR] [MEASURE],
int S, int M)
{
    int i; // індекс рядка ( номер датчика )
    int j; // індекс стовпця ( номер вимірювання )
    int sum; // сума вимірювань для обрахунку середнього значення
              // для кожного вимірювання на основі показників
              // трьох датчиків

    for( j = 0; j < M; j++ ){
        sum = 0;
        for( i = 0; i < S; i++ )
            sum += data[i][j];

        mMea[j] = (float)sum / (float)S;
    }

}

//-----

```

```

void printMeanForMeasure( const float mMea [MEASURE], int M )
{
    int i; // індекс елемента

    printf("\n\n\n");
    printf("-----\n");
    printf("Середні значення для кожного із вимірювань:");

    for( i = 0; i < M; i++ )
        printf("\nВим. [%d]: %7.2f", i, mMea[i] );

    printf("\n");
}

```

На рис. 6.7 показано результат виконання приведеної програми, що носить демонстраційний характер для ілюстрації роботи із двовимірними масивами.

```

Датчик[0]:   Вим.[0]   Вим.[1]   Вим.[2]   Вим.[3]   Вим.[4]   Вим.[5]
Датчик[0]:       1         2         4         5         2         1
Датчик[1]:       5         3         1         2         9         8
Датчик[2]:       1         5         3         2         4         2

-----
Середні значення вимірювань для кожного датчика:
Датчик[0]: 2.50
Датчик[1]: 4.67
Датчик[2]: 2.83

-----
Середні значення для кожного із вимірювань:
Вим.[0]: 2.33
Вим.[1]: 3.33
Вим.[2]: 2.67
Вим.[3]: 3.00
Вим.[4]: 5.00
Вим.[5]: 3.67

```

Рис. 6.7 — Результати роботи програми по обробці двовимірного статичного масиву

Якщо одновимірний статичний масив можна інтерпретувати як вектор-рядок, двовимірний статичний масив можна для зручності аналізу розглядати як матрицю або також двовимірний масив можна розглядати як одновимірний масив, кожен елемент якого є вектор-рядок (тобто кожен елемент якого є одновимірний масив), то трьохвимірний статичний масив можна розглядати як масив матриць, або масив таблиць (такий варіант інтерпретації трьохвимірного

масиву може застосуватися для зручності представлення масиву) [3].

Наприклад, якщо оголошений такий трьохвимірний статичний масив типу int:

```
#define K 2
#define N 3
#define M 4

int b[K][N][M];
```

Масив **b** може розглядатися як масив, що складається із **K** елементів, де кожен елемент представляє собою двовимірний масив розміром **N×M** (тобто, іншими словами можна сказати, що масив складається із **K** матриць(таблиць) де кожна матриця має розмір **N×M** елементів). Для того, що виконати обробку всіх елементів такого трьохвимірного масиву, можна, наприклад, використовувати три циклічні оператори — в першому операторі циклу буде виконуватися зміна індексу (номеру) матриці, у вкладеному операторі циклу буде виконуватися зміна індексу рядка матриці, а в операторі циклу, який вкладений у попередній цикл — буде виконуватися зміна індексу стовпця матриці. Таким чином можна виконати обробку всіх елементів масиву, змінюючи відповідні індекси в дозволених межах [3].

Нижче приводиться програма, в якій виконується ініціалізація трьохвимірного статичного масиву, а також виконується виведення значень елементів масиву в консольне вікно, використовуючи для цього функцію, яка отримує в якості своїх параметрів масив та значення відповідних вимірів:

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

#define K 2
#define N 3
#define M 4

// прототип функції
void print3Darray(const int [K][N][M], int matrixNo, int row, int col);
```

```

int main()
{
    int b[K][N][M] = {
        {
            { 5, 1, 3, 7 },
            { 4, 6, 8, 2 },
            { 6, 2, 7, 4 }
        },
        {
            { 3, 6, 9, 2 },
            { 8, 5, 1, 8 },
            { 6, 3, 5, 2 }
        }
    };

    setlocale(LC_CTYPE, "Ukrainian");

    print3Darray( b, K, N, M);

    return 0;
}

//-----

void print3Darray(const int b[K][N][M], int matrixNo, int row, int col)
{
    int n; // індекс відповідає за номер матриці: n = 0...matrixNo-1
    int r; // індекс рядка матриці: r = 0...row-1
    int c; // індекс стовпця матриці: c = 0...col-1

    printf("\n\n");

    for( n = 0; n < matrixNo; n++) {

        printf("\nМатриця номер %d:\n", n);

        for( r = 0; r < row; r++ ){
            for( c = 0; c < col; c++ )
                printf("%3d", b[n][r][c] );
            printf("\n");
        }
    }
}

```

На рис. 6.8 показано результат виконання програми. Схематичне представлення трьохвимірного статичного масиву показано на рис. 6.9, відповідно до якого для зручності трьохвимірний масив інтерпретується як масив матриць.

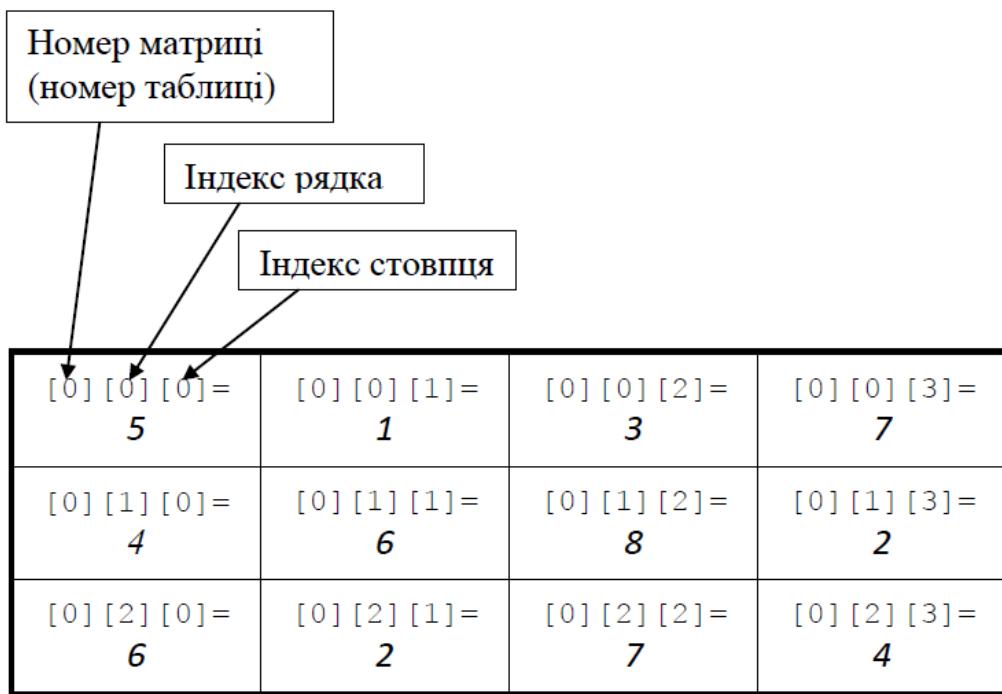
```

Матриця номер 0:
5 1 3 7
4 6 8 2
6 2 7 4

Матриця номер 1:
3 6 9 2
8 5 1 8
6 3 5 2

```

Рис. 6.8 — Виведення елементів трьохвимірного масиву у вигляді послідовності матриць



$[1][0][0] = 3$	$[1][0][1] = 6$	$[1][0][2] = 9$	$[1][0][3] = 2$
$[1][1][0] = 8$	$[1][1][1] = 5$	$[1][1][2] = 1$	$[1][1][3] = 8$
$[1][2][0] = 6$	$[1][2][1] = 3$	$[1][2][2] = 5$	$[1][2][3] = 2$

Рис. 6.9 — Інтерпретація трьохвимірного масиву як набір матриць

Чотирьохвимірний масив f , який має 4 виміри, оголошується, наприклад, таким чином [5]:

```
#define L 5
#define K 2
#define N 3
#define M 4

int f[L][K][N][M];
```

Масив f може розглядатися як масив, що складається із L елементів, де кожен елемент представляє собою трьохвимірний масив (тобто набір матриць із K штук, де кожна матриця має розмір $N \times M$). Така інтерпретація дещо штучна, але вона дозволяє більш просто оперувати багатовимірними масивами.

6.3 Питання для самоконтролю

1. Що таке масив? Яке значення індексу у першого елемента одновимірного масиву?
2. Чим відрізняється одновимірний масив від змінної?
3. Одновимірний масив має розмір 25. Яке значення індексу в останнього елементу масиву?
4. Яким чином можна виконати ініціалізацію всіх елементів масиву значенням нуль?
5. Напишіть функцію, яка в якості параметра отримує масив та кількість елементів в масиві, і в ході своєї роботи присвоює кожному елементу масиву значення його індексу.
6. Напишіть функцію, яка в якості параметрів отримує двовимірний масив та розміри масиву і обраховує добуток всіх елементів масиву.
7. Що представляє собою ім'я масиву в контексті вказівників?
8. Чи можна в статичному масиві змінювати розміри масиву на етапі виконання програми?
9. Як визначити кількість байтів пам'яті, що виділяється для зберігання елементів статичного масиву?

7. ОДНОВИМІРНІ ТА БАГАТОВИМІРНІ ДИНАМІЧНІ МАСИВИ

Статичні масиви характеризуються тим, що розмір масиву необхідно визначати на етапі написання програми. Розмір статичного масиву — це константне значення. Але можуть виникати задачі, коли розмір масиву повинен задаватися в процесі роботи програми. Причому заздалегідь невідомо яку кількість елементів повинен містити такий масив. В таких випадках виникає необхідність використання динамічних масивів. При роботі із динамічними масивами необхідний об'єм пам'яті для масиву повинен виділятися під час виконання програми. Таким чином, потрібно визначати необхідний розмір для масиву (наприклад, задати це значення з клавіатури або обрахувати в деякому виразі, або прочитати з деякого файлу), після цього виконати виклик функції, яка дозволяє виділити пам'ять для динамічного масиву, і якщо пам'ять буде виділена — функція поверне вказівник (адресу) на перший байт виділеного блоку пам'яті. Таким чином, при роботі із динамічними масивами потрібно використовувати вказівники. Однак, може так трапитися, що пам'ять не буде виділена під динамічний масив, в такому випадку відповідна функція по виділенню пам'яті повертає значення `NULL`. Якщо функція повернула значення `NULL` — це значить, що динамічний масив не був створений. Тому при роботі з динамічними масивами потрібно завжди виконувати перевірку значення, що повертає функція по виділенню пам'яті, і якщо це значення дорівнює `NULL`, — то передбачити відповідні дії, наприклад, проінформувати про це користувача і завершити програму, або спробувати створити динамічний масив з меншою кількістю елементів. Після того як динамічний масив використаний у відповідному алгоритмі чи відповідній процедурі обробки даних, і у ньому немає необхідності при подальшій роботі програми, — необхідно звільнити пам'ять, яка була виділена для динамічного масиву [3, 4, 6].

Для виконання дій, що пов'язані із виділенням та звільненням пам'яті для динамічного масиву використовуються наступні функції: `calloc()`, `malloc()`,

`realloc()` та `free()`. Для того, щоб скористатися цими функціями необхідно підключити заголовочний файл `stdlib.h`.

Функція `calloc()` має такий прототип [5]:

```
void * calloc( size_t Num_Of_Elements, size_t Size_Of_Elements );
```

Функція повинна отримувати два параметри: перший параметр — це кількість елементів в динамічному масиві; другий параметр — це розмір в байтах елементу масиву. Таким чином, задаючи кількість елементів в масиві та об'єм пам'яті, який потрібен для зберігання одного елемента масиву, функція в разі успіху виділяє сукупний необхідний об'єм пам'яті та повертає адресу на перший байт виділеного блоку пам'яті (тип даних результату, що функція повертає — це вказівник на `void` (вказівник `void *`), — це значить, що функція повертає адресу). Надалі, значення адреси присвоюється відповідному вказівнику, і цей вказівник виступає в ролі імені масиву. Після цього робота із динамічним масивом може проводитися як із звичайним статичним масивом. Якщо пам'ять не була виділена — функція повертає значення `NULL`.

При передачі у функцію значення розміру елемента масиву в байтах доречно вказувати не константне значення, наприклад, 1, 2, 4 або 8 (в залежності який тип даних обрано для динамічного масиву), а визначати відповідний розмір в байтах, використовуючи операцію `sizeof`, оскільки розміри деяких типів даних можуть різнятися в залежності від системи. Використання операції `sizeof` дозволить підвищити якість програмного коду і дозволить спростити процес внесення змін в програмний код, в разі необхідності переходу на іншу систему.

Особливість роботи функції `calloc()` полягає в тому, що в разі успішного виділення пам'яті під динамічний масив, елементи динамічного масиву стають рівними нулю.

Інша функція, яка дозволяє виконувати виділення пам'яті — це функція `malloc()`.

Функція `malloc()` має такий прототип [5]:

```
void * malloc(size_t Size_of_Memory_Block);
```

Функція `malloc()` в якості параметра отримує один параметр — це сукупний об'єм пам'яті, який необхідно виділити для динамічного масиву. Значення цього параметра може бути обчислене як добуток кількості елементів динамічного масиву на розмір в байтах одного елементу масиву. Analogічно попередній функції, при визначенні розміру в байтах одного елемента масиву, рекомендується використовувати операцію `sizeof`. У випадку успіху — функція `malloc()` повертає адресу першого байту виділеного блоку пам'яті (вказівник `void *`). У випадку, якщо пам'ять не була виділена — функція повертає значення `NULL`. На відміну від функції `calloc()`, функція `malloc()` не обнулює значення в пам'яті, яка була виділена під динамічний масив, тому значення елементів динамічного масиву вважаються невизначеними.

Функція `realloc()` має такий прототип [5]:

```
void * realloc(void * Memory_Block, size_t New_Size_of_Mem_Block);
```

Функція `realloc()` використовується для зміни розміру раніше виділеного блоку пам'яті. Перший параметр зберігає адресу (першого байту) виділеного блоку пам'яті, другий параметр визначає розмір нового блоку пам'яті в байтах. Функція повертає адресу блоку пам'яті зміненого розміру. Якщо об'єм пам'яті недостатній для збільшення розміру блоку пам'яті, то функція повертає значення `NULL`. Якщо значення первого параметра — це адреса першого байту (адреса початку) раніше виділеного блоку пам'яті, а другий параметр дорівнює 0, то функція `realloc()` звільняє блок пам'яті, адреса початку якого зберігалася в першому параметру, і функція повертає значення `NULL`. Якщо перший параметр буде дорівнювати `NULL`, а другий параметр буде визначати розмір в байтах блоку пам'яті необхідного об'єму, то функція `realloc()` в разі успіху виділить пам'ять і поверне адресу на початок цього блоку пам'яті, а якщо не вдасться виділити пам'ять — поверне значення `NULL`.

Крім функцій виділення пам'яті використовується також функція, що звільняє раніше виділену пам'ять для динамічного масиву. Для використовується функція `free()`.

Функція `free()` має такий прототип [5]:

```
void free(void * Memory_Block);
```

Функція в якості параметра отримує адресу початку блоку пам'яті, що до цього був виділений за допомогою функції `calloc()`, `malloc()` або `realloc()`. Важливо не забувати звільнити пам'ять, яка була виділена під динамічний масив, якщо в програмі не передбачається подальше використання відповідного динамічного масиву.

7.1 Одновимірний динамічний масив. Особливості роботи із одновимірними динамічними масивами

В якості прикладу роботи із одновимірними динамічними масивами буде розглянуто динамічний масив типу `char`, `int` та `double`. Враховуючи, що при роботі із динамічними масивами виникає необхідність виділяти пам'ять для відповідного масиву та зберігати адресу початку блоку пам'яті, то при роботі із динамічним масивом, який призначений для зберігання значень типу `char` необхідно в програмі оголосити вказівник на `char` — такому вказівнику буде присвоюватися адресу блоку пам'яті, в якому будуть зберігатися значення типу `char`. Аналогічно, необхідно буде оголосити вказівник на `int` та вказівник на `double`.

Нехай всі три динамічні масиви типу `char`, `int` та `double` мають однакову кількість елементів (однаковий розмір), наприклад, 3 і це значення зберігається в змінній:

```
int SizeOfDynArray = 3;
```

Нехай, оголошено наступні вказівники, які, по суті, будуть виступати в якості імен динамічних масивів:

```
char * pArrCh;  
int * pArrInt;  
double * pArrDbl;
```

Імена вказівників — це ідентифікатори, тому при виборі імен потрібно притримуватися правил складання ідентифікаторів. Бажано обирати імена, які б містили певну інформацію про призначення масиву, оскільки такі імена дозволяють легше виконувати аналіз тексту програми та в разі необхідності робити відповідні зміни та правки в коді.

Для виділення пам'яті буде застосовано функцію `calloc()`, також буде виконано перевірку на предмет виділення пам'яті, і якщо функція `calloc()` поверне значення `NULL`, то передбачено завершення програми. Вказані дії реалізуються таким чином:

```
pArrCh = (char *) calloc( SizeOfDynArray, sizeof( char ) );  
pArrInt = (int *) calloc( SizeOfDynArray, sizeof( int ) );  
pArrDbl = (double *) calloc( SizeOfDynArray, sizeof( double ) );  
  
if( pArrCh==NULL || pArrInt==NULL || pArrDbl==NULL ) {  
    printf( "\nУвага!" );  
    printf( "\nНе вдалося виділити пам'ять для динамічного масиву." );  
    printf( "\nПрограму буде завершено. " );  
    exit(0);  
}
```

Враховуючи, що функція `calloc()` повертає вказівник на `void`, то при присвоєнні значення, що повертається функцією, відповідному вказівнику, застосується операція приведення типу (перетворення типу), — `(char *)`, `(int *)` та `(double *)`.

Схематично виділені блоки пам'яті, а також значення вказівників `pArrCh`, `pArrInt`, `pArrDbl`, які вказують на перші байти відповідних блоків пам'яті, показано на рис. 7.1.

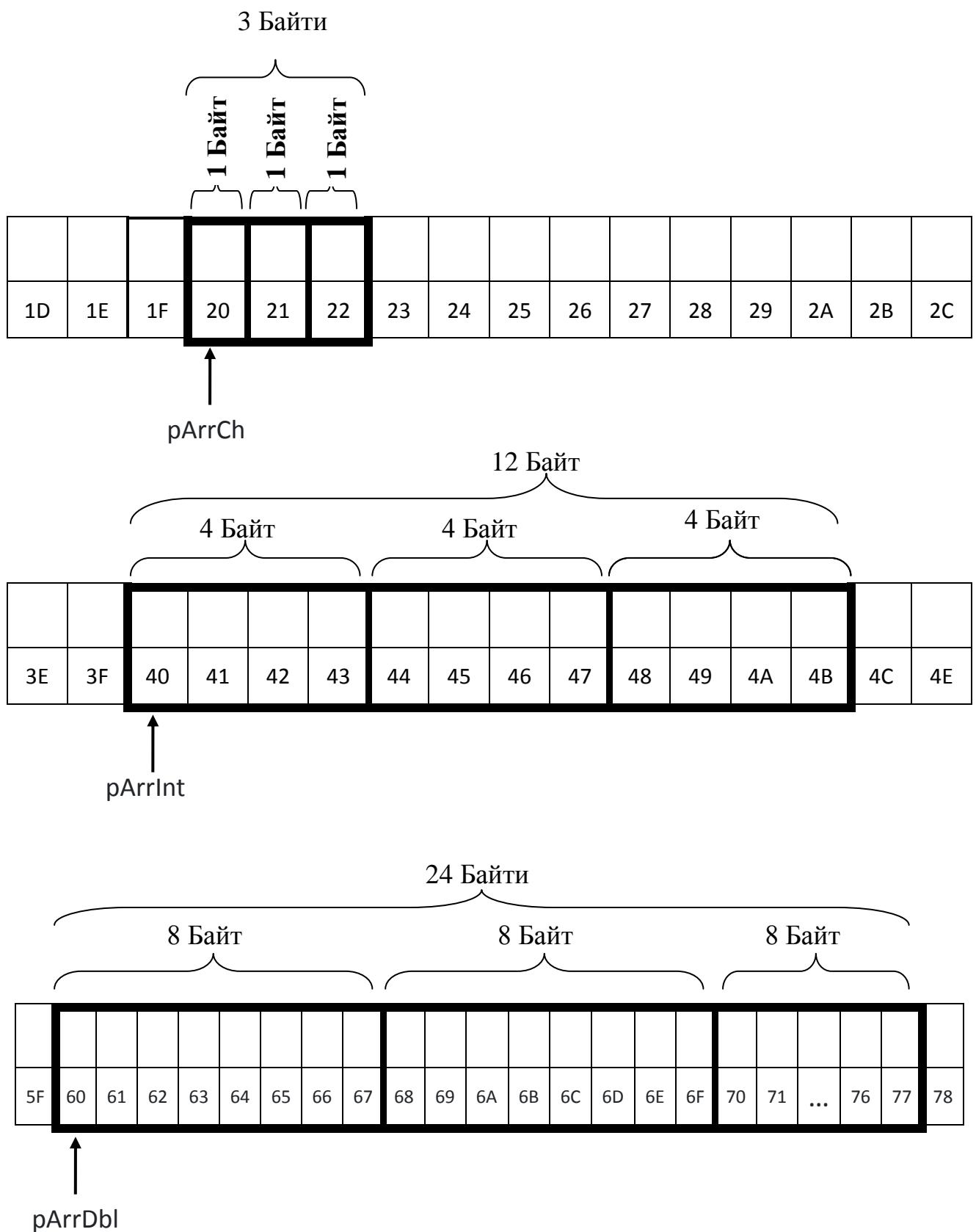


Рис. 7.1 — Схематичне розміщення блоків пам'яті для динамічних масивів, що складаються із 3-х елементів, типу даних `char`, `int` та `double` відповідно

Для наведеного прикладу вказівник pArrInt отримує значення адреси 20, — це адреса першого байту виділеного блоку пам'яті. Ця адреса співпадає із адресою першого елемента масиву (тип даних `char` займає 1 байт пам'яті, тому блок пам'яті для збереження трьох значень типу `char` буде потребувати 3-х байтів).

Вказівник pArrInt має значення 40, — це адреса першого байту виділеного блоку пам'яті розміром 12 байт (оскільки одне значення типу `int` займає 4 байти пам'яті, а всього елементів в масиві — 3). Адреса, яка зберігається у вказівнику pArrInt буде також відповідати адресі першого елемента динамічного масиву типу `int`.

Вказівник pArrDbl має значення 60, — це адреса першого байту виділеного блоку пам'яті розміром 24 байти (оскільки одне значення типу `double` займає 8 байт пам'яті, а всього елементів в масиві — 3). Разом з тим, ця адреса співпадає також із адресою першого елемента масиву.

Таким чином, оскільки значення вказівника дорівнює адресі першого елемента масиву, то для того, щоб дістатися до значення першого елементу масиву, необхідно застосувати операцію розіменування. Наступні дії дозволяють присвоїти першим елементам відповідних масивів значення 5, 312 та -5.125:

```
*pArrCh = 5;  
*pArrInt = 312;  
*pArrDbl = -5.125;
```

Для того, щоб перейти до адреси наступного елемента масиву потрібно до вказівника додати 1, цю величину називають *зміщенням*. Результат додавання буде визначатися типом даних. При додаванні 1 до вказівника pArrCh значення адреси буде збільшуватися на 1 байт. При додаванні 1 до вказівника pArrInt значення адреси буде збільшуватися на 4 байти. При додаванні 1 до вказівника pArrDbl значення адреси буде збільшуватися на 8 байт. Тобто, результат додавання (віднімання) до вказівника певного цілого числа призводить до

збільшення (зменшення) значення вказівника на величина яка визначається як результат добутку значення доданку (значення зміщення) на кількість байт, яка виділяється для відповідного типу даних. З урахування цього, другому елементу відповідного масиву можна присвоїти значення шляхом зміщення до даного елемента масиву, а також при цьому необхідно застосувати операцію розіменування до отриманої адреси. Наприклад, нехай другі елементи відповідних масивів отримують значення 72, 498 та 12.75:

```
* (pArrCh+1) = 72;  
* (pArrInt+1) = 498;  
* (pArrDbl+1) = 12.75;
```

Для того, щоб перейти до третього елемента масиву, потрібно до вказівника додати зміщення 2, отримати адресу відповідного елемента, а застосувавши операцію «*» виконати доступ до значення елемента, і, наприклад, присвоїти певну числову величину. Нехай треті елементи відповідних масивів отримують значення -4, -128 та 99.5:

```
* (pArrCh+2) = -4;  
* (pArrInt+2) = -128;  
* (pArrDbl+2) = 99.5;
```

Таким чином, даний підхід дозволяє розглядати значення вказівника як початкову точку (перший елемент масиву), відносно якої необхідно зсунутися на деяку кількість елементів, щоб дістатися до потрібного елементу масиву. Якщо величину зміщення задати у вигляді змінної, яка в циклі змінюється від 0 до значення, що на одиницю менше ніж розмір масиву, можна виконати обробку всіх елементів масиву. Фактично, зміщення виступає в якості індексу елементу. Тому при роботі із динамічними масивами можна використовувати не тільки формат запису, який передбачає використання операції розіменування до обрахованої адреси, але також застосовувати формат запису, що передбачає запис імені вказівника та відповідного індексу елемента масиву. Як було зазначено в попередньому розділі, ім'я статичного масиву — це вказівник на перший елемент. В даному випадку робота також ведеться із вказівником, який

отримує адресу блоку пам'яті на етапі її виділення, і отримана адреса відповідає адресі першого елемента, тому імена вказівників `pArrCh`, `pArrInt` та `pArrDbl` можна також використовувати як імена масивів, і тим самим використовувати відповідний формат запису для звернення до елементів масиву через індекс елементу. Тому, наступні записи є еквівалентними [4, 7]:

```
// Значення 1го елементу масиву
*pArrCh == *(pArrCh + 0) == pArrCh[0] == 5
*pArrInt == *(pArrInt + 0) == pArrInt[0] == 312
*pArrDbl == *(pArrDbl + 0) == pArrDbl[0] == -5.125

// Значення 2го елементу масиву
*(pArrCh + 1) == pArrCh[1] == 72
*(pArrInt + 1) == pArrInt[1] == 498
*(pArrDbl + 1) == pArrDbl[1] == 12.75

// Значення 3го елементу масиву
*(pArrCh + 2) == pArrCh[2] == -4
*(pArrInt + 2) == pArrInt[2] == -128
*(pArrDbl + 2) == pArrDbl[2] == 99.5
```

З урахуванням зазначеного, схематично розміщення елементів масивів та значення відповідних вказівників зображено на рис. 7.2.

При створенні функції по обробці одновимірного динамічного масиву в якості параметрів, які необхідно передавати у функцію, виступають вказівник (який може грати роль імені динамічного масиву) та розмір динамічного масиву. Якщо передбачається, що функція не повинна змінювати значення елементів масиву, то за допомогою службового слова `const`, записаним перед типом даних вказівника у прототипі функції та в заголовку функції при її описі, можна заборонити функції змінювати значення елементів масиву, оскільки функція буде сприймати такий динамічний масив як константий, а змінювати значення констант в процесі роботи програми — заборонено.

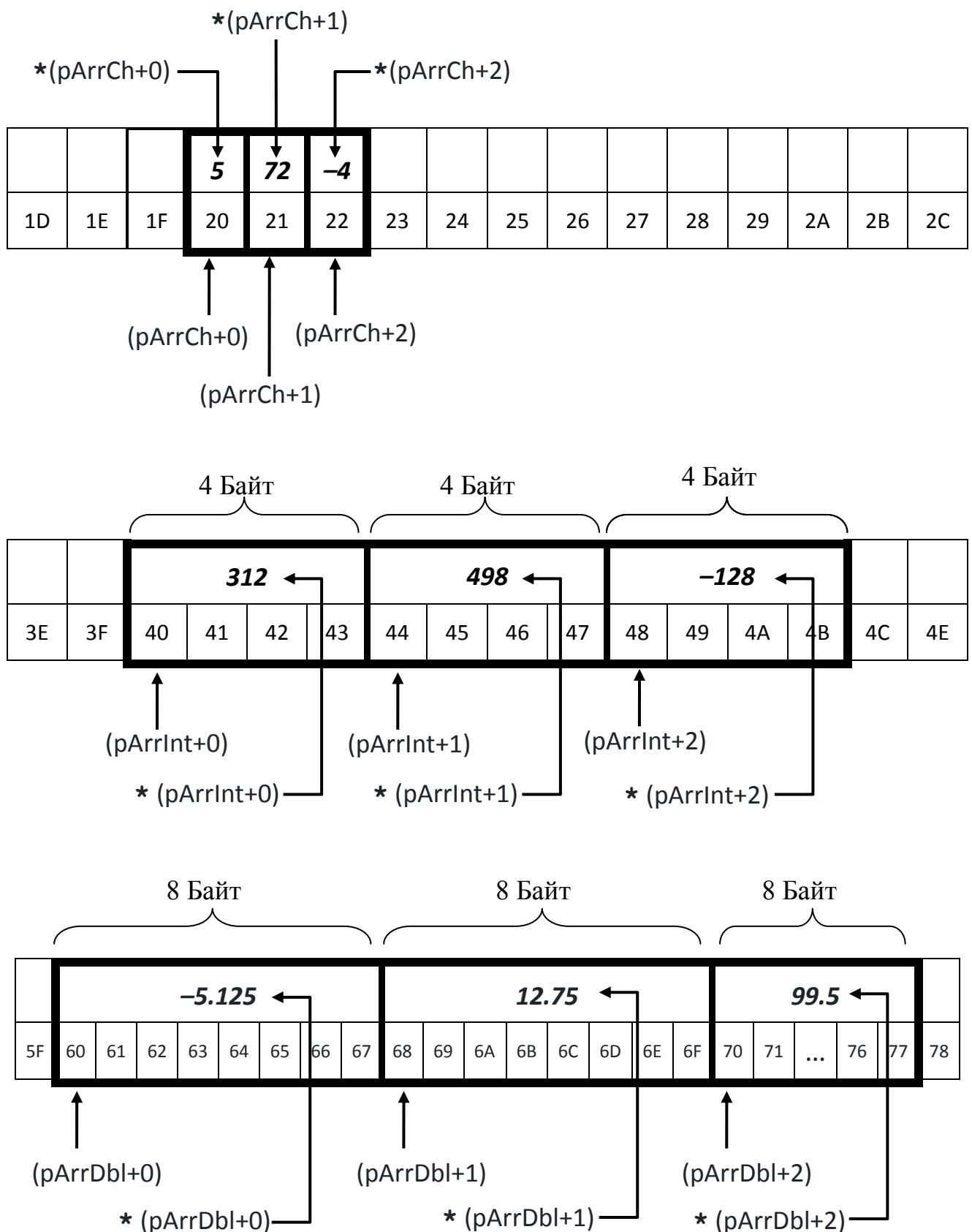


Рис. 7.2 — Схематичне розміщення елементів динамічних масивів різних типів даних в пам'яті комп'ютера

Нижче приведено приклад роботи із динамічним масивом. В програмі створюється одновимірний динамічний масив типу float. Розмір масиву вводиться користувачем з клавіатури. Значення елементів масиву генеруються як випадкові величини із діапазону 0.0...1.0 — ця задача покладена на функцію genRandValueFloatArray(). Функція printFloatDynArray() використовується для виведення значень масиву на екран в консольне вікно. Також реалізована функція meanValueInFloatArray(), яка виконує підрахунок середнього значення в масиві:

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

void genRandValueFloatArray( float * floatArr, int SizeOfDynArray );
void printFloatDynArray( const float * floatArr, int SizeOfDynArray );
float meanValueInFloatArray( const float * floatArr, int SizeOfDynArray );

int main()
{
    float * pFloatArr;
    int SizeOfDynArray;
    float meanFloatValue;

    setlocale( LC_CTYPE, "Ukrainian" );
    printf("\nВведіть значення кількості елементів в масиві.");
    printf("\nЯкщо значення МЕНШЕ ніж 1 – програма завершиться.");
    printf("\n\nК-сть елементів: ");
    scanf("%d", &SizeOfDynArray);

    if( SizeOfDynArray < 1 ){
        printf("\n\nНекоректні вхідні дані!!! ");
        printf("\nВихід з програми.\n\n");
        exit(0);
    }

    pFloatArr = (float *)calloc( SizeOfDynArray, sizeof( float ) );
    if( pFloatArr == NULL ){
        printf("\n\n Увага!!!! ");
        printf("\nПам'ять для динамічного масиву не виділена!");
        printf("\nЗавершення виконання програми.");
        exit(0);
    }
}
```

```

genRandValueFloatArray( pFloatArr, SizeOfDynArray );

printFloatDynArray( pFloatArr, SizeOfDynArray );

meanFloatValue = meanValueInFloatArray( pFloatArr, SizeOfDynArray );

printf("\n\n-----");
printf("\nСереднє значення в масиві: %.2f\n", meanFloatValue );
return 0;
}

//----- Опис Функцій -----
void genRandValueFloatArray( float * floatArr, int SizeOfDynArray )
{
    int index;

    for( index = 0; index < SizeOfDynArray; index++ )
        *(floatArr + index) = (float)rand() / (float)RAND_MAX;
}

//-----

void printFloatDynArray( const float * floatArr, int SizeOfDynArray )
{
    int index;

    printf("\n\n\nЗначення елементів масиву:\n\n");
    for( index = 0; index < SizeOfDynArray; index++ )
        printf("%.2f ", *(floatArr + index) );
}

//-----

float meanValueInFloatArray( const float * floatArr, int SizeOfDynArray )
{
    int index;
    float sum = 0.0;

    for( index = 0; index < SizeOfDynArray; index++ )
        sum += floatArr[index];

    return sum/SizeOfDynArray;
}

```

На рис. 7.3 показано результат виконання програми.

```
Введіть значення кількості елементів в масиві.  
Якщо значення МЕНШЕ ніж 1 – програма завершиться.  
К-сть елементів: 10  
  
Значення елементів масиву:  
0.00 0.56 0.19 0.81 0.59 0.48 0.35 0.90 0.82 0.75  
  
Середнє значення в масиві: 0.54
```

Рис. 7.3 — Результат обробки даних в одновимірному динамічному масиві

7.2 Багатовимірні динамічні масиви. Особливості роботи із багатовимірними динамічними масивами

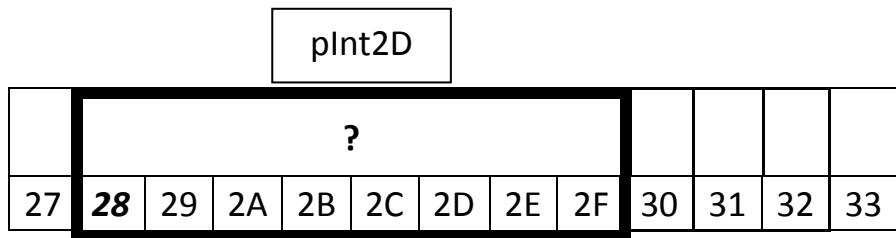
Крім одновимірних динамічних масивів при вирішенні відповідних задач може виникати необхідність у використанні багатовимірних динамічних масивів, наприклад, двовимірних, трьохвимірних і т.д. Розглянемо принцип роботи із багатовимірними динамічними масивами на прикладі двовимірного динамічного масиву. Основна ідея, яка лежить при виконанні виділення пам'яті для двовимірного динамічного масиву може бути розповсюджена і на масиви, що мають більшу кількість вимірів, ніж 2.

Якщо при роботі із одновимірним динамічним масивом доводилося працювати із вказівником відповідного типу, то при роботі із двовимірним динамічним масивом, необхідно використовувати вказівник на вказівник. Тип даних вказівника на вказівник обирається в залежності від того, який тип даних обрано для елементів масиву. Наприклад, нехай необхідно в програмі створити двовимірний динамічний масив для зберігання значень типу даних `int`, — в такому випадку необхідно створити відповідний вказівник на вказівник, нехай в якості прикладу ім'я вказівника обрано `pInt2D`, і цей вказівник оголошується в програмі наступним чином:

```
int ** pInt2D;
```

Цей вказівник призначений зберігати адресу блоку пам'яті, в якій будуть зберігатися адреси рядків матриці (двовимірний масив для простоти інтерпретації буде розглядатися як матриця). Кількість байтів, яка виділяється для зберігання значень вказівників залежить від системи. Наприклад, в 64-х розрядній системі Windows, для зберігання значення адреси виділяється 8 байт.

Для зберігання значення вказівнику `pInt2D` виділяється 8 байт пам'яті. На даному етапі вказівник лише оголошений, але не ініціалізований, тому невідомо чому він дорівнює. Схематично покажемо розміщення в пам'яті даного вказівника та його значення (яке поки що невідомо) таким чином:



В даному прикладі, адреса блоку пам'яті, де зберігається значення вказівника `pInt2D`, дорівнює 28_{16} — значення адреси традиційно записується в шістнадцятковій системі числення. Для того, щоб визначити адресу блоку пам'яті, де зберігається вказівник `pInt2D`, необхідно застосувати операцію «&». Для представленого схематичного варіанту розміщення вказівника `pInt2D` в пам'яті, будемо мати: $\&pInt2D = 28_{16}$. Для того щоб отримати значення, яке зберігається за відповідною адресою, потрібно до адреси застосувати операцію «*»: $*(\&pInt2D) = ?$ — оскільки вказівник не ініціалізований, то невідомо його початкове значення. До значення вказівника також можна дістатися по його імені, тому:

$$*(\&pInt2D) = pInt2D = ?$$

Вказівнику `pInt2D` буде надаватися певне значення в процесі роботи програми.

Для збереження значення розмірів динамічного масиву можуть використовуватися змінні, значення яких задаються з клавіатури або визначаються іншим чином в процесі роботи програми. Нехай для двовимірного динамічного масиву значення кількості рядків та кількості стовпців зберігаються в змінних `numberOfRows` та `numberOfColumns`, яким присвоєно такі значення:

```
int numberOfRows = 2;  
int numberOfColumns = 3;
```

Наступний крок — це створити одновимірний масив, який буде призначений для зберігання адрес блоків пам'яті рядків масиву (адрес рядків матриці). В даному прикладі передбачається, що двовимірний масив буде складатися із двох рядків. Тому потрібно створити одновимірний масив із двох елементів (кількість елементів відповідає кількості рядків — це значення зберігається в змінній `numberOfRows`), в кожному елементі буде зберігатися адреса блоку пам'яті відповідного рядка: в першому елементі одновимірного масиву буде зберігатися адреса першого рядка матриці, в другому елементі одновимірного масиву — буде зберігатися адреса другого рядка матриці. Для виділення пам'яті використовується функція `calloc()`. Якщо пам'ять не буде виділена для одновимірного масиву для зберігання адрес рядків матриці, — програма завершується:

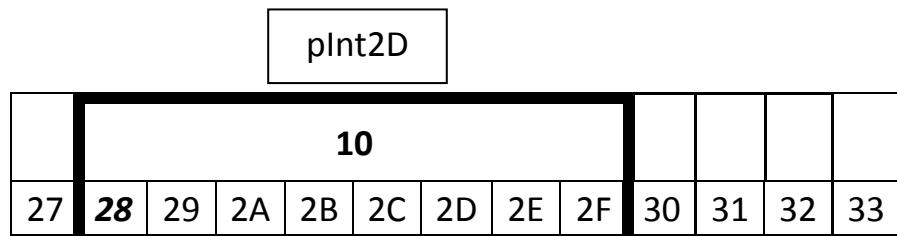
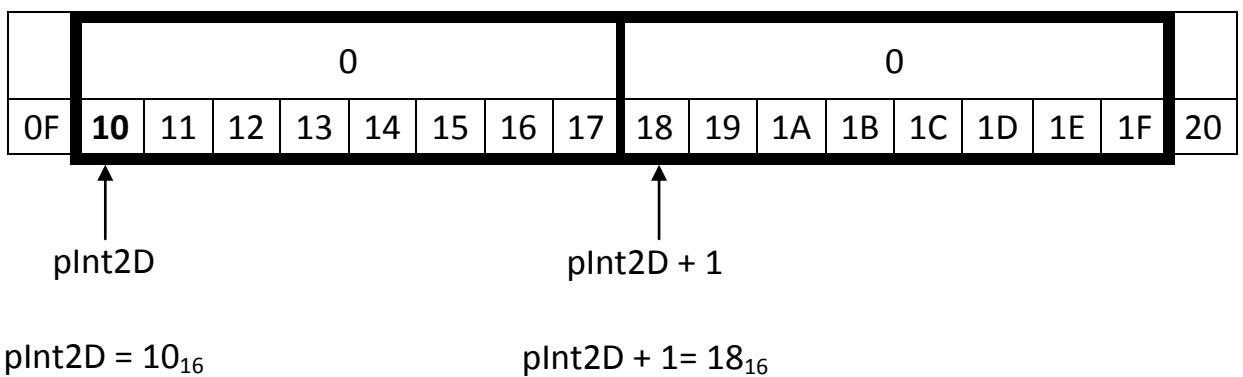
```
pInt2D = (int **)calloc( numberOfRows, sizeof( int* ) );  
  
if(pInt2D == NULL ) {  
    printf("\n\n Увага!!!!");  
    printf("\nПам'ять для динамічного масиву не виділена!");  
    printf("\nЗавершення виконання програми.");  
    exit(0);  
}
```

При присвоєнні вказівнику `pInt2D` адреси, що повертається функцією `calloc()`, виконується явне приведення типів (перетворення типів). Після

виконання даного фрагменту коду, вказівник отримує адресу блоку пам'яті, яка була виділена для зберігання адрес рядків матриці.

Нехай вказівник `pInt2D` отримав значення адреси 10_{16} , що відповідає адресі блоку пам'яті, за якою буде зберігатися адреса першого рядка матриці. Шляхом додавання 1 до значення вказівника `pInt2D` відбувається зміщення до адреси блоку пам'яті другого елемента масиву, в якому буде зберігатися адреса другого рядка матриці, тобто $\text{pInt2D} + 1 = 18_{16}$.

Схематично це можна відобразити таким чином:



На наступному кроці необхідно першому елементу (елементу з адресою 10_{16}) присвоїти значення адреси блоку пам'яті, де будуть зберігатися значення *першого* рядка матриці. А другому елементу (елементу з адресою 18_{16}) присвоїти значення адреси блоку пам'яті, де будуть зберігатися значення *другого* рядка матриці. Для того, щоб мати доступ до значень елементів масиву, призначеного для зберігання адрес — до вказівника `pInt2D` застосовується операція «`*`», а також застосовується зміщення (0 — для першого елемента, 1 — для другого елемента). Програмний код, який реалізує описану процедуру представлено нижче:

```

for( i = 0; i < numberOfRows; i++ ) {

    * (pInt2D+i) = (int *)calloc( numberOfColumns, sizeof( int ) );

    if( pInt2D == NULL ) {
        printf( "\n\n Увага!!!!" );
        printf( "\nПам'ять для динамічного масиву не виділена!" );
        printf( "\nЗавершення виконання програми." );
        exit(0);
    }
}

```

Потрібно звернути увагу на те, що в даному фрагменті коду виконується виділення пам'яті для одновимірних масивів для зберігання не адрес, а значень типу `int`, тому при виклику функції `calloc()` в другому параметрі записано `sizeof(int)`.

В циклі змінна `i` зберігає поточне значення зміщення для переходу до відповідного елементу масиву, якому присвоюється значення адреси блоку пам'яті, яка виділена для зберігання рядка матриці. Оскільки кількість рядків в розглянутому прикладі зберігається в змінній `numberOfRows`, то ця змінна використовується в операторі циклу `for`. Так само ця змінна зберігає кількість елементів для одновимірного масиву, який був створений на попередньому кроці для зберігання адрес блоків пам'яті рядків матриці.

У виклику функції `calloc()` використовується змінна `numberOfColumns`, оскільки ця змінна зберігає кількість елементів в рядку матриці.

Схематично, описана процедура представлена на рис. 7.4, де показані відповідні масиви, які були створені на кожному етапі — спочатку був створений одновимірний масив для зберігання адрес, а потім створено два рядки (два одновимірні масиви), в кожному з яких зберігається 3 значення цілого типу (типу `int`), і адреси цих рядків були присвоєні елементам одновимірному масиву, який зберігає адреси. Тепер, використовуючи зміщення і розіменування або використовуючи індексацію (ім'я вказівника в цьому

випадку грає роль імені масиву) можна дістатися до потрібного елементу масиву.

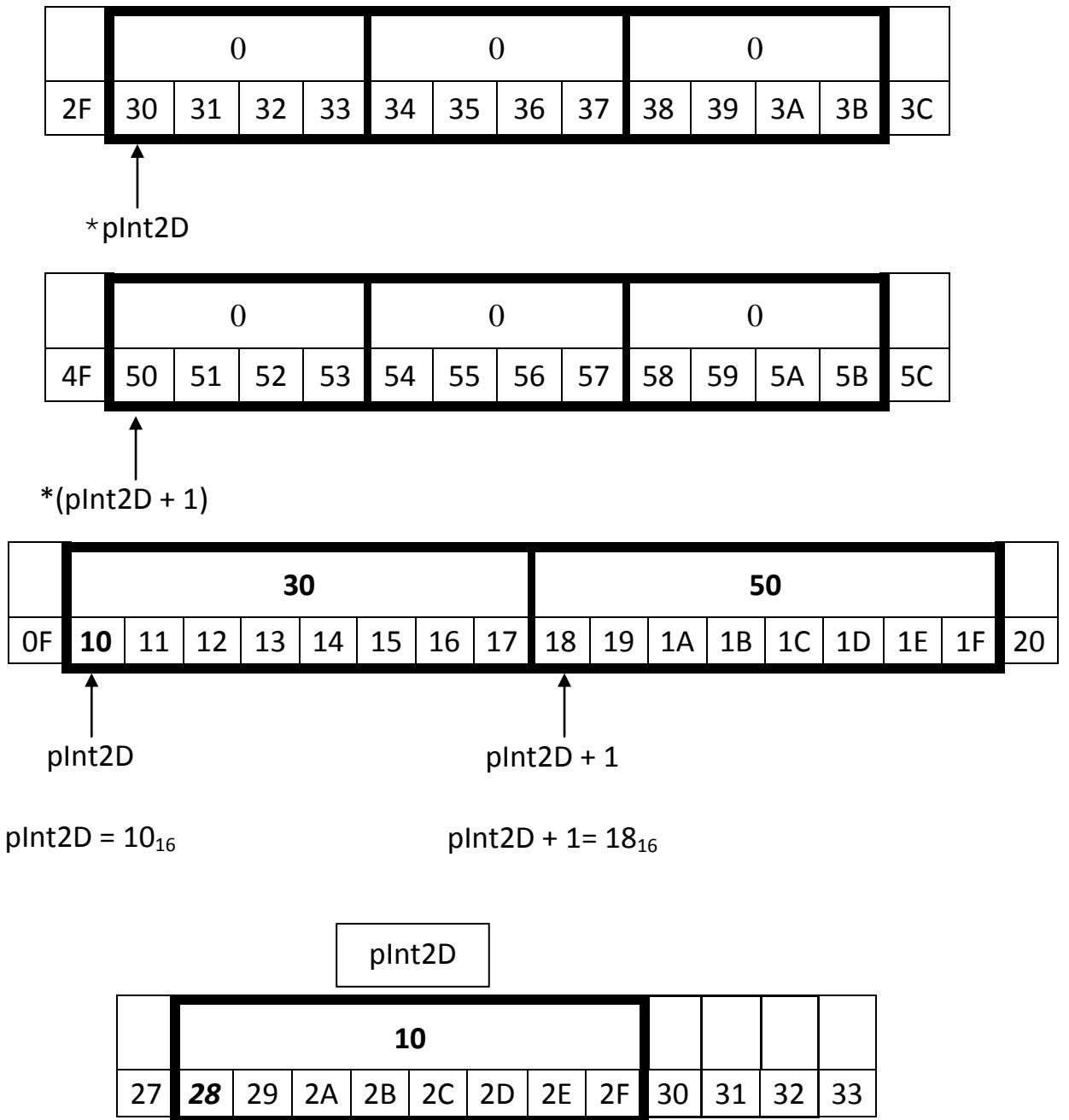


Рис. 7.4 — Розміщення в пам'яті двовимірного динамічного масиву типу int

Нижче приведено програмний код, який містить розглянуті процедури, а також виконується виведення відповідних адрес на екран в консольне вікно з метою вивчення особливостей роботи із вказівниками в контексті двовимірних

динамічних масивів. В кінці програми виконується звільнення пам'яті, яка виділялась для масиву:

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

int main()
{
    int numberOfRows = 2;
    int numberOfColumns = 3;
    int i;
    int j;

    int ** pInt2D;

    setlocale( LC_CTYPE, "Ukrainian" );

    pInt2D = (int **)calloc(numberOfRows, sizeof(int*) );
    if(pInt2D == NULL ){
        printf("\n\n Увага!!!\n");
        printf("\nПам'ять для динамічного масиву не виділена!\n");
        printf("\nЗавершення виконання програми.\n");
        exit(0);
    }

    for( i = 0; i < numberOfRows; i++ ){
        *(pInt2D+i) = (int *)calloc( numberOfColumns, sizeof(int) );

        if( *(pInt2D+i) == NULL ){
            printf("\n\n Увага!!!\n");
            printf("\nПам'ять для динамічного масиву не виділена!\n");
            printf("\nЗавершення виконання програми.\n");
            exit(0);
        }
    }

    printf("\n\n");

    printf("&pInt2D = %p", &pInt2D );
    printf("\n\n");
    printf("Адреси елементів масиву, в яких зберігаються адреси рядків:\n");
```

```

for( i = 0; i < numberOfRows; i++ )
    printf("\n (pInt2D+%d) = %p", i, (pInt2D+i) );

printf("\n\n");
printf("Адреси рядків:\n");
for( i = 0; i < numberOfRows; i++ )
    printf("\n * (pInt2D+%d) = %p", i, *(pInt2D+i) );

printf("\n\n");
for( i = 0; i < numberOfRows; i++ ){
    printf("\n\nАдреси елементів %d-го рядка:\n", i);
    for( j = 0; j < numberOfColumns; j++ )
        printf("\n (* (pInt2D+%d)+%d) = %p", i, j, (* (pInt2D+i)+j) );
}

printf("\n\n");

printf("\n\n");
printf("\n Введення значень елементів масиву:\n");
for(i = 0; i < numberOfRows; i++ )
    for( j = 0; j < numberOfColumns; j++ ){
        printf("[%d] [%d]=", i, j );
        scanf("%d", &pInt2D[i][j] );
    }

printf("\n\n");
printf("\n Масив:\n");
for(i = 0; i < numberOfRows; i++ ){
    for( j = 0; j < numberOfColumns; j++ )
        printf("%4d", *(pInt2D+i)+j );

    printf("\n");
}

for( i = 0; i < numberOfRows; i++ )
    free( pInt2D[i] );
free( pInt2D );

return 0;
}

```

На рис. 7.5 показано результат виконання програми.

```

&pInt2D = 0000000000022FE28
Адреси елементів масиву, в яких зберігаються адреси рядків:
<pInt2D+0> = 00000000000535C10
<pInt2D+1> = 00000000000535C18

Адреси рядків:
*<pInt2D+0> = 00000000000535C30
*<pInt2D+1> = 00000000000535C50

Адреси елементів 0-го рядка:
<*<pInt2D+0>+0> = 00000000000535C30
<*<pInt2D+0>+1> = 00000000000535C34
<*<pInt2D+0>+2> = 00000000000535C38

Адреси елементів 1-го рядка:
<*<pInt2D+1>+0> = 00000000000535C50
<*<pInt2D+1>+1> = 00000000000535C54
<*<pInt2D+1>+2> = 00000000000535C58

Введення значень елементів масиву:
[0][0]=-2
[0][1]=5
[0][2]=9
[1][0]=3
[1][1]=-1
[1][2]=7

Масив:
-2 5 9
 3 -1 7

```

Рис. 7.5 — Результати виконання програми

Показане на рис. 7.6 схематичне розміщення відповідних елементів масиву в пам'яті, також демонструє зв'язок між операцією розіменування та індексацією в контексті роботи із масивами. Для компактності запису ім'я вказівника на вказівник було обрано p . Результати, які показані на рис. 7.5 та рис. 7.6, дають змогу співставити теоретичні аспекти та практичну реалізацію щодо питання організації роботи та використання двовимірних динамічних масивів.

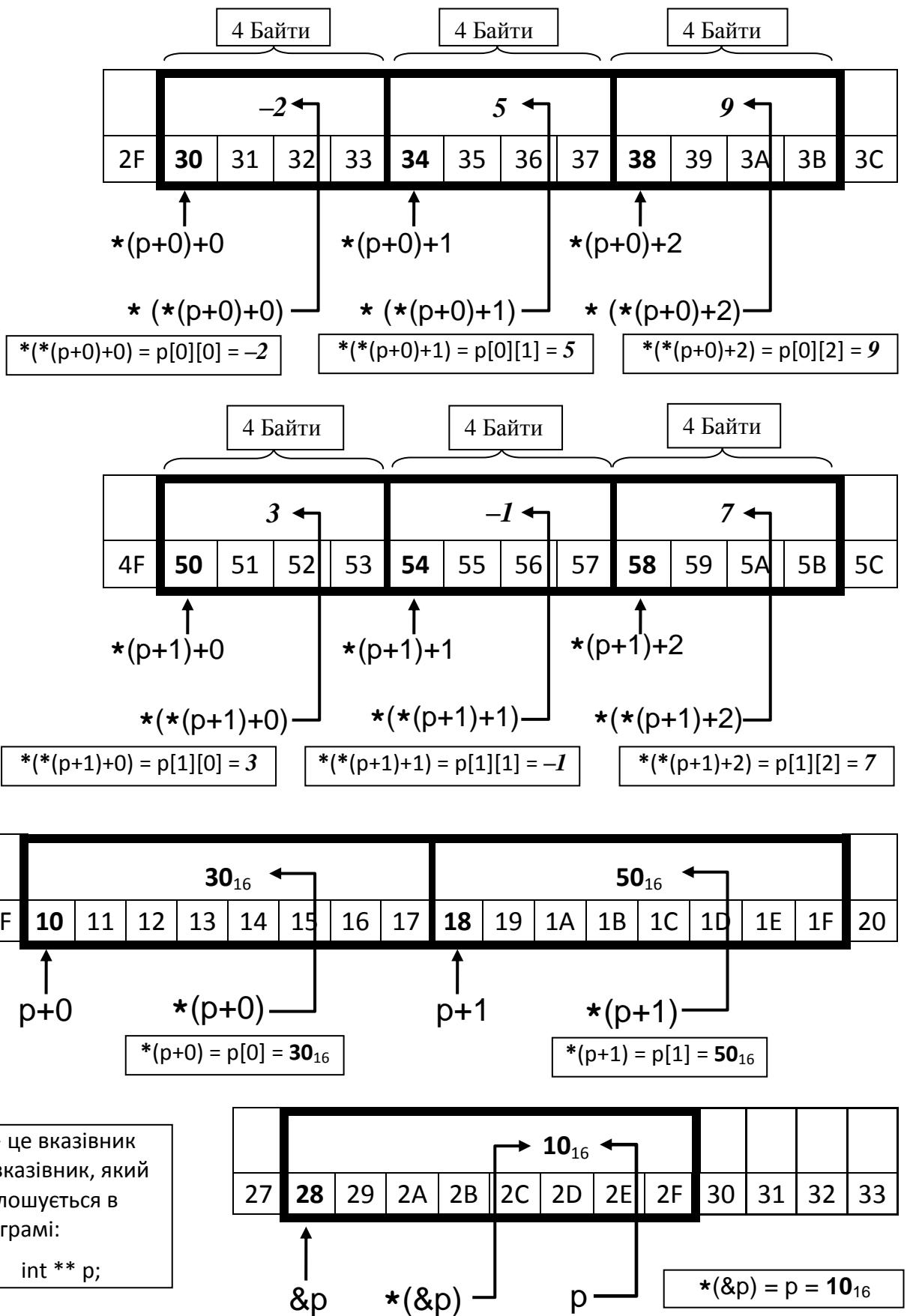


Рис. 7.6 — Схема зберігання двовимірного динамічного масиву типу int

Нижче приводиться приклад роботи із двовимірним динамічним масивом типу float. В програмі значення елементів масиву задаються з клавіатури, відбувається підрахунок суми від'ємних елементів масиву, результати відображаються на екрані в консольному вікні. Для реалізації вказаних дій в програмі написані відповідні функції:

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

void getNumberOfRowsColums( int *, int * );
float ** getMemoryFor2DArray( int R, int C );
void initFloat2DArray( float ** arrName, int R, int C );
void printFloat2DArray( float ** arrName, int R, int C );
float sumNegative( float ** arrName, int R, int C );
void printResult( float sumOfNegativElems );

int main()
{
    float ** pFlt2D;
    int numberOfRows;
    int numberOfColumns;
    float sum;
    int i;

    setlocale(LC_CTYPE, "Ukrainian");

    getNumberOfRowsColums( &numberOfRows, &numberOfColumns );

    pFlt2D = getMemoryFor2DArray( numberOfRows, numberOfColumns );

    initFloat2DArray( pFlt2D, numberOfRows, numberOfColumns );

    printFloat2DArray( pFlt2D, numberOfRows, numberOfColumns );

    sum = sumNegative( pFlt2D, numberOfRows, numberOfColumns );

    printResult( sum );

    for( i = 0; i < numberOfRows; i++ )
        free( pFlt2D[i] );
    free( pFlt2D );

    return 0;
}
```

```

//----- Опис Функцій -----
void getNumberOfRowsColumns( int * pRows, int * pColumns )
{
    printf("\n\nВведіть значення кількості рядків та стовпців в масиві.");
    printf("\nЯкщо введені значення МЕНШІ ніж 1 - програма завершується");

    printf("\n\nК-сть рядків: ");
    scanf("%d", pRows );

    if( *pRows < 1 )
        exit(0);

    printf("\nК-сть стовпців: ");
    scanf("%d", pColumns );

    if( *pColumns < 1 )
        exit(0);
}

//-----

float ** getMemoryFor2DArray( int R, int C )
{
    int i;
    float ** tempFlt2D;

    tempFlt2D = (float **)calloc( R, sizeof( float * ) );

    if( tempFlt2D == NULL ){
        printf("\n\n Увага!!! ");
        printf("\nПам'ять для динамічного масиву не виділена!");
        printf("\nЗавершення виконання програми.");
        exit(0);
    }

    for( i = 0; i < R; i++ ){
        tempFlt2D[i] = (float *)calloc( C, sizeof( float ) );

        if( tempFlt2D[i] == NULL ){
            printf("\n\n Увага!!! ");
            printf("\nПам'ять для динамічного масиву не виділена!");
            printf("\nЗавершення виконання програми.");
            exit(0);
        }
    }

    return tempFlt2D;
}

```

```

//-----
void initFloat2DArray( float ** Flt2D, int R, int C )
{
    float tempFloatValue;
    int i, j;

    printf("\n\nВведіть значення елементів масиву:\n\n");

    for( i = 0; i < R; i++ )
        for( j = 0; j < C; j++ ){
            printf("[%d] [%d]=", i, j );
            scanf("%f", &tempFloatValue );
            Flt2D[i][j] = tempFloatValue;
        }
}

//-----

void printFloat2DArray( float ** Flt2D, int R, int C )
{
    int i, j;

    printf("\n\nМасив:\n\n");

    for( i = 0; i < R; i++ ){
        for( j = 0; j < C; j++ )
            printf("%8.2f", Flt2D[i][j] );
        printf("\n");
    }
}

//-----

float sumNegative( float ** Flt2D, int R, int C )
{
    int i, j;
    float tempSum;

    tempSum = 0.0;

    for( i = 0; i < R; i++ )
        for( j = 0; j < C; j++ )
            if( Flt2D[i][j] < 0 )
                tempSum += Flt2D[i][j];

    return tempSum;
}

```

```

//-----

void printResult( float sumOfNegativeElements )
{
    printf("\n\n");
    printf("-----\n");
    printf("\nСума від'ємних значень в масиві: \n");
    printf(" %7.2f \n\n", sumOfNegativeElements );
}

```

На рис. 7.7 показано результат виконання програми.

```

Введіть значення кількості рядків та стовпців в масиві.
Якщо введені значення МЕНШІ ніж 1 – програма завершується

К-сть рядків: 2
К-сть стовпців: 4

Введіть значення елементів масиву:

[[0][0]=-0.25
[[0][1]=1.75
[[0][2]=2.0
[[0][3]=4.2
[[1][0]=13.9
[[1][1]=2.45
[[1][2]=-10.5
[[1][3]=0.45

Масив:
 -0.25    1.75    2.00    4.20
 13.90    2.45   -10.50    0.45

-----
Сума від'ємних значень в масиві:
 -10.75

```

Рис. 7.7 — Результат виконання програми

Описані процедури можуть бути використані для роботи з масивами, що мають більшу кількість вимірів. В цьому випадку ідея залишається тією ж, але реалізація може суттєво ускладнюватися, наприклад, якщо потрібно створити динамічний трьохвимірний масив для збереження значень типу `double`, то необхідно в програмі оголосити такий вказівник:

```
double *** pointerName;
```

де `pointerName` — це ім'я вказівника (будь-який правильно складений ідентифікатор). І в подальшому потрібно виконувати виділення пам'яті для відповідних масивів по принципу, який був описаний вище.

7.3 Питання для самоконтролю

1. Поясніть яким чином виконується виділення пам'яті для динамічного одновимірного масиву. Наведіть приклад програмної реалізації виділення пам'яті для одновимірного динамічного масиву типу `double`.
2. Яка різниця між функціями `calloc()` та `malloc()`?
3. Навіщо застосовується функція `free()`?
4. Напишіть функцію, яка в якості параметрів отримує одновимірний динамічний масив і знаходить суму елементів масиву, які мають непарні індекси.
5. Поясніть яким чином виконується виділення пам'яті для двовимірного динамічного масиву. Наведіть приклад програмної реалізації створення двовимірного динамічного масиву, використовуючи для цього функцію `calloc()` та `malloc()`.
6. Яке значення повертається функціями `calloc()` та `malloc()` у випадку, якщо не вдалося виділити вказаний об'єм пам'яті?
7. Як за допомогою операції `sizeof` визначити який об'єм пам'яті виділяється в системі для зберігання значень типу `char`, `short`, `int`, `long`, `float`, `double`, `long double` ?
8. Яким чином змінюється значення вказівника при додаванні або відніманні деякого цілого числа?
9. Яка різниця між динамічним та статичним одновимірним (дтовимірним) масивом?
10. Напишіть фрагмент коду по виділенню пам'яті для трьохвимірного динамічного масиву типу даних `double`.

8. РОБОТА З ФАЙЛАМИ. СИМВОЛЬНІ РЯДКИ ТА ФУНКЦІЇ ПО РОБОТІ ІЗ СИМВОЛЬНИМИ РЯДКАМИ

Організація введення та виведення даних може будуватися на [7]:

1. потоковому (високорівневому) введенні/виведенні;
2. низькорівневому введенні/виведенні;
3. введенні/виведенні на рівні портів пристройв.

Для забезпечення потокового введення/виведення в мові програмування С передбачені відповідні функції стандартної бібліотеки.

Особливості низькорівневого введення/виведення можуть визначатися відповідною операційною системою під управлінням якої функціонує електронно-обчислювальна машина. Враховуючи, що використання функцій низькорівневого введення/виведення робить відповідну програму системозалежною, стандартом мови С визначається високорівневе (потокове) введення/виведення.

При потоковому введенні/виведенні передача даних відбувається у вигляді послідовності байтів. При цьому введення/виведення є буферизованим. Інформація передається у вигляді блоків даних відповідного розміру, а не по одному байту. Буферизація дозволяє пришвидшити передачу даних при роботі з файлами та забезпечує мінімізацію звертання до фізичного пристрою, який, наприклад, виступає в якості джерела інформації або отримувача інформації.

При запуску програми автоматично відкриваються наступні потоки [3, 4]:

1. `stdin` — стандартний потік введення;
2. `stdout` — стандартний потік виведення;
3. `stderr` — стандартний потік виведення повідомлення про помилки.

За замовчуванням, стандартному потоку введення `stdin` ставиться у відповідність клавіатура. Стандартному потоку виведення `stdout` ставиться у відповідність екран монітору. Стандартному потоку виведення помилок `stderr` також зазвичай ставиться у відповідність екран монітору.

8.1. Особливості роботи із файлами. Види файлів

При роботі із файлами існують декілька режимів відкриття файлу. Необхідний режим потрібно вказувати при відкритті файлу (або при створенні файлу, якщо файлу не існує).

Файл може бути відкритий в текстовому режимі або в двійковому (бінарному) режимі. В бінарному режимі доступний кожен байт, що записаний у файлі. Двійковий режим може бути застосований для запису у файл даних в тому виді, в якому вони зберігаються в пам'яті комп'ютера [3, 4, 7].

При відкритті файлу в текстовому режимі використовуються такі режими [3—7, 9]:

`r` — відкриття текстового файлу для читання даних;

`w` — відкриття текстового файлу для запису. Якщо файл не існує на диску, то він буде створений. Якщо файл існує і в ньому міститься інформація, то вміст файла стирається;

`a` — відкриття текстового файлу для додавання інформації. Нові дані додаються в кінець файла. Якщо файл не існує, то він буде створений, і інформація буде записана в створений файл;

`r+` — відкриття текстового файла для читання та запису даних. Файл, що відкривається в даному режимі, повинен існувати. Інформацію з файла можна читати, а також записувати, однак заборонено збільшення розміру файла;

`w+` — відкриття текстового файла для запису та читання. Якщо файл, який відривається існує на диску, то його вміст стирається. Допускається запис даних та читання в будь-якому місці файла. Дозволяється збільшення розміру файла в процесі дозапису інформації в процесі роботи із файлом. Якщо файл не існує, то він буде створений;

`a+` — відкриття текстового файла для читання та запису, причому читати можна весь файл, записувати нові дані тільки в кінець файла. При

відкритті існуючого файлу в даному режимі, вміст файлу повинен бути збережений на відміну від режиму `w+`. Якщо файл відсутній він повинен створитися.

Ці ж режими також застосовуються і при відкритті файлу в двійковому (бінарному) режимі, при цьому до назви відповідного режиму додається літера `b`: `rb`, `wb`, `ab`, `r+b` або `rb+`, `w+b` або `wb+`, `a+b` або `ab+`.

Для відкриття файлу використовується функція `fopen()`, яка має наступний прототип [5, 7]:

```
FILE * fopen(const char * Name_of_File, const char * ModeOpen);
```

Перший параметр — `Name_of_File`, — це ім'я файлу, — в якості цього параметру може бути використаний символічний рядок, в якому зберігається ім'я файлу (в імені файлу зазначається також розширення файлу). Другий параметр — `ModeOpen`, — це також символічний рядок, який містить режим відкриття файлу. Перелік режимів було зазначено вище. Результат, що функція `fopen()` повертає, — це вказівник на структуру `FILE`. У випадку, якщо функція `fopen()` не змогла відкрити файл, або не змогла створити файл, то вона повертає значення `NULL`, таким чином це значення може сигналізувати про те, що виникли проблеми при намаганні відкрити файл, і в такому разі необхідно передбачити подальші дії, наприклад, завершити програму, або зробити запит користувачу щодо введення з клавіатури імені іншого файла. Таким чином, при роботі із файлами, в програмі необхідно створити вказівник типу `FILE *`, і за допомогою такого вказівника буде виконуватися звернення до файла з метою прочитання інформації з файла або запису інформації у файл.

По завершенню роботи з файлом — необхідно закрити відповідні потоки, які використовувалися для роботи із файлом. Для цього використовується функція `fclose()`. Функція має наступний прототип [5, 7]:

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

Параметр `ptrFile` — це вказівник, який був пов'язаний із файлом, з яким проводилась робота. Функція `fclose()` — це функція, що повертає результат типу `int`. Якщо функція відпрацювала успішно, вона повертає значення 0, якщо функція не змогла закрити потік, вона повертає значення EOF. Константа EOF визначається за допомогою директиви препроцесора в заголовочному файлі `stdio.h`. Цілий ряд функцій, які призначені для роботи із потоками (в тому числі із файлами), при досягненні кінця файлу повертають значення EOF (абревіатура EOF — End Of File). Числове значення константи EOF зазвичай дорівнює `-1`, але краще використовувати саме константу EOF, щоб програма була переносима, і могла працювати на інших системах, де, наприклад, EOF може відрізнятися від значення `-1`.

При виконанні дій щодо зчитування даних з файлу і для запису даних у файл можуть використовуватися різні функції, зокрема функції `fscanf()` та `fprintf()`, які мають такі прототипи [5]:

```
int fscanf(FILE * ptrFile, const char * Format, ...)

int fprintf(FILE * ptrFile, const char * Format, ...);
```

Дані функції по принципу своєї роботи відповідають функціям `scanf()` та `printf()`, за виключенням того, що в якості першого параметра обидві функції — `fscanf()` та `fprintf()`, — повинні отримувати вказівник на потік. При роботі із файлом це повинен бути вказівник на структуру `FILE`, який використовується в програмі для взаємодії із файлом на диску. Обидві ці функції (так само як і функції `scanf()` та `printf()`) повертають результат типу `int`.

Функції `fscanf()` та `scanf()` повертають число, що відповідає кількості аргументів, яким успішно були присвоєні значення. Якщо досягнуто кінець файлу, то повертається значення EOF, це ж значення повертається якщо виникла помилка до першого присвоєння значення параметру, що зчитується з файлу або клавіатури відповідно. Якщо повертається значення 0, це вказує що присвоєння параметру значення, яке зчитувалося з файлу чи клавіатури, не відбулося. Таким чином, значення, що повертається функцією `fscanf()` або

функцією `scanf()` можна використовувати при аналізі успішності зчитування даних з файлу або з клавіатури.

Функції `fprintf()` та `printf()` повертають значення, що дорівнює кількості успішно виведених символів (у файл або на екран в консольне вікно відповідно). Якщо при цьому виникли помилки — повертається від'ємне значення.

Крім функцій `fscanf()` та `fprintf()` також можуть застосовуватися інші функції, зокрема `getc()`, `putc()`, `fgets()`, `fputs()` та ін.

Функція `getc()` призначена для зчитування одиночного символу з потоку. Прототип функції наступний [5, 7]:

```
int getc(FILE * ptrFile);
```

Функція в якості параметра отримує вказівник на структуру `FILE` — потік введення, який був зв'язаний з відповідним файлом при виклику функції `fopen()`.

Функція `getc()` повертає значення типу `int`, яке відповідає коду символу із кодової таблиці. Якщо функція в процесі зчитування символів з файлу досягла кінця файла, то вона повертає значення `EOF`, таким чином це дає змогу виконувати відповідні перевірки щодо досягнення кінця файла.

Функція `putc()` використовується для запису символів у файл. Прототип функції `putc()` такий [5, 7]:

```
int putc(int Symbol, FILE * ptrFile);
```

В якості першого параметру задається символ, який виводиться в потік введення, другий параметр — це вказівник на потік (вказівник на структуру `FILE` при запису символа у файл). Функція `putc()` повертає значення типу `int`. У випадку успішного запису символу — функція повертає код цього символу. У випадку помилки, що виникла при записі символу, — функція `putc()` повертає значення `EOF`.

Для зчитування символного рядка використовується функція `fgets()`. Ця функція наступний прототип [5, 7]:

```
char * fgets(char * Buffer, int MaxCountOfSymbols, FILE * ptrFile);
```

Перший параметр функції — це вказівник на `char`, тобто по суті перший параметр — це ім'я масиву, куди буде записуватися символний рядок. Другий параметр — визначає максимальне значення довжини символного рядка (тобто задає розмір символного рядка), який може бути безпечно збережений у заданому масиві, сюди ж включається і нульовий символ (символ `\0`), який виступає в якості признаку кінця символного рядку. Тобто, якщо задати значення параметру `MaxCountOfSymbols` рівним 10, то при умові, що розмір масиву `Buffer`, наприклад, більший або рівний 10, то при зчитуванні з файлу символів (наприклад, букви, цифри, розділові знаки) в масив буде збережено 9 символів прочитаних з файлу, а 10-м символом запишеться символ `\0`. Третій параметр — це вказівник на структуру `FILE`, тобто, вказує на потік введення (звідки виконується зчитування даних). При успішному зчитуванні даних, функція повертає вказівник на `char` — тобто вказівник на символний рядок (адреса блоку пам'яті, де збережений символний рядок). Якщо виникла помилка в ході виконання зчитування даних із файла або якщо було досягнуто кінець файла, то функція `fgets()` повертає значення `NULL`.

Для запису символного рядка у файл, використовується функція `fputs()`. Прототип даної функції такий [5, 7]:

```
int fputs(const char * strArray, FILE * ptrFile);
```

Перший параметр функції — це адреса символного рядка, який буде записуватися у файл (наприклад, на позиції первого параметра може записуватися ім'я відповідного масиву, що зберігає символний рядок, оскільки ім'я масиву — це і є адреса). Другий параметр — це вказівник на структуру

`FILE`. У випадку помилки в ході запису символного рядка у файл, функція `fputs()` повертає значення `EOF`.

При зчитуванні даних з файлу може виникнути необхідність переміщатися по файлу. Файл може розглядатися як послідовність байтів даних, і в ході роботи програми може з'являтися потреба переміститися до певного байту або переміститися на певну кількість позицій відносного поточної позиції у файлі. Для цих цілей можуть використовуватися функції `fseek()`, `ftell()`, `fgetpos()` та `fsetpos()`, `rewind()`.

Функція `fseek()` дозволяє встановлювати або зміщувати показчик поточної позиції у файлі, дозволяючи, наприклад, виконувати зчитування даних із файла, починаючи із певної заданої позиції (з певного байту). В цьому випадку файл може розглядатися як масив символів, де кожен символ має розмір 1 байт. Функція дозволяє зміститися до певного байту, щоб, наприклад, за допомогою відповідної функції виконувати зчитування даних з деякої позиції, а не обов'язково з самого початку файла. Ця функція може виконати зміщення показчика поточної позиції в кінець файла, з метою, наприклад, почати зчитування даних з кінця файла, або виконати зміщення відносно поточної позиції на відповідну кількість байтів. Функція має такий прототип [5, 7]:

```
int fseek(FILE * ptrFile, long Offset, int Start);
```

Перший параметр — це вказівник на потік (вказівник на структуру `FILE`, який використовується для роботи із файлом на диску). Другий параметр — це зміщення. Зміщення визначає на скільки позицій (на скільки байт) потрібно зміститися відносно поточної позиції у файлі. Зміщення може бути як додатним та і від'ємним. Якщо зміщення має від'ємне значення, то відбувається зміщення в напрямку початку файла відносно поточної позиції на задану кількість байтів. Якщо значення зміщення додатне, то відбувається зміщення в напрямку кінця файла відносно поточної позиції. Третій параметр визначає позицію у файлі,

відносно якої відбувається зміщення. Третій параметр (`int Start`) може встановлюватися шляхом запису для нього відповідного значення за допомогою таких констант [3, 4]:

SEEK_SET	Початок файлу
SEEK_CUR	Поточна позиція у файлі
SEEK_END	Кінець файлу

Якщо функція `fseek()` успішно відпрацювала, то вона повертає значення 0. Якщо в ході роботи функції виникли помилки, то функція повертає ненульове значення.

Інша функція `ftell()` — призначена для визначення кількості байтів від початку файлу для поточної позиції у файлі. Ця функція має наступний прототип [5]:

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

В якості параметра функція отримує вказівник на потік (вказівник на структуру `FILE`) і повертає поточне значення показника позиції у файлі, який відкритий в програмі, причому це значення поточної позиції визначається як кількість байтів від початку файлу, при цьому нумерація байтів у файлі починається з нуля. Данна функція орієнтована на роботу із файлами, що відкриті в бінарному режимі, і при роботі із файлами, що відкриті в текстовому режимі, може повертати значення позиції у файлі, що не відповідає дійсності.

Функція `fgetpos()` може використовуватися при роботі із файлами великих розмірів. Прототип такої функції має вигляд [5]:

```
int fgetpos(FILE * ptrFile , fpos_t * Position);
```

Дана функція зберігає значення показника поточної позиції у файлі в комірці пам'яті, адреса якої зберігається у вказівнику `Position`. Тип даних `fpos_t` може визначатися на основі базового типу даних. Розглядаючи прототип функції, видно, що вона повертає ціле значення типу `int`. Якщо функція

`fgetpos()` відпрацювала без помилок, то вона повертає значення 0. Якщо виникли помилки — функція `fgetpos()` повертає ненульове значення.

Функція `fsetpos()` встановлює показник положення позиції у файлі. Прототип функції має такий вигляд [5]:

```
int fsetpos(FILE * ptrFile, const fpos_t * Position);
```

Значення позиції, яку потрібно встановити, зберігається в комірці пам'яті, адреса якої зберігається у вказівнику `Position`. Якщо функція відпрацювала без помилок, вона повертає значення 0. Якщо виникли помилки, функція `fsetpos()` повертає ненульове значення.

Функція `rewind()` дозволяє переміститися на початок файлу [4]. Прототип даної функції такий [5]:

```
void rewind(FILE * ptrFile);
```

Функція `rewind()` отримує лише вказівник на структуру `FILE`. Функція має тип `void`, тобто вона не повертає результат.

Ряд функцій по роботі із файлами повертають значення `EOF` як признак досягнення кінця файлу, а також як признак появи помилки. В такому випадку розмежувати яка саме подія відбувалася виключно на основі значення, що повернула відповідна функцію, неможливо. В такій ситуації на допомогу може прийти функція `feof()`. Функція `feof()` має такий прототип [5, 7]:

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

Функція `feof()` дозволяє виконувати контроль досягнення кінця файлу при зчитування даних з файлу. Інша функція — `ferror()` дозволяє контролювати наявність помилок при роботі функцій по зчитуванню даних з файлу або при записі даних у файл [5, 7]:

```
int ferror(FILE * ptrFile);
```

Функція повертає ненульове значення, якщо виникли помилки при роботі функцій по запису або зчитуванню даних. Якщо помилок немає — повертається 0.

Для запису у файл даних в двійковому коді (у вигляді в якому дані зберігаються в пам'яті комп'ютера) використовується функція `fwrite()`. При цьому відповідний файл необхідно відкривати (створювати) в двійковому (бінарному) режимі. Функція `fwrite()` має такий прототип [5, 7]:

```
size_t fwrite(const void * Str, size_t Size, size_t Count, FILE * ptrFile);
```

Функція отримує чотири параметри. Перший параметр містить адресу пам'яті де зберігаються дані, які необхідно записати у файл. Другий параметр — це розмір в байтах тієї порції даних, яка буде записуватися у файл. Третій параметр — це кількість таких порцій даних, які будуть записуватися у файл за один виклик функції `fwrite()`. Четвертий параметр — вказівник на структуру `FILE`, який використовується для роботи із відкритим файлом. За допомогою функції `fwrite()` за один її виклик можна, наприклад, записати у файл одне значення — значення елементу масиву, значення змінної тощо. А можна, наприклад, одним викликом записати декілька елементів масиву, які розміщаються в пам'яті послідовно — вказавши у першому параметрі адресу початку блоку пам'яті, де зберігаються відповідні дані, в другому параметрі зазначивши розмір в байтах того типу даних, який використаний для зберігання відповідних значень, а в третьому параметрі вказати кількість таких даних.

Для зчитування двійкових даних із файлу використовується функція `fread()` [3, 4]. Функція має такий прототип [5, 7]:

```
size_t fread(void * Buffer, size_t ElemSize, size_t Count, FILE * ptrFile);
```

Призначення параметрів аналогічні функції `fwrite()`. Тільки в даному випадку перший параметр — це значення адреси блоку пам'яті, куди будуть записуватися двійкові дані, які читаються із двійкового файлу. За допомогою функції за один виклик можна зчитати одну порцію даних, або ж прочитати з файла та зберегти, наприклад, в масив декілька порцій даних.

Нижче приведені приклади використання відповідних функцій по зчитуванню даних з файла та запису даних у файл.

```

#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<locale.h>

int main()
{
    FILE * pTestInputFile;
    int singleLetter;

    setlocale(LC_CTYPE, "Ukrainian");

    pTestInputFile = fopen( "input_text_data.txt", "r" );

    if( pTestInputFile == NULL ) {
        printf("\nПомилка при спробі відкрити файл!!!!");
        printf("\nНатисніть будь-яку клавішу для завершення");
        getch();
        exit(EXIT_FAILURE);
    }

    singleLetter = getc( pTestInputFile );
    while ( !(feof( pTestInputFile ) ) ) {
        putc( singleLetter, stdout );
        singleLetter = getc( pTestInputFile );
    }

    putc('\n', stdout);

    fclose( pTestInputFile );
}

return 0;
}

```

В програмі відкривається текстовий файл, який був створений в текстовому редакторі «Блокнот», файл має ім'я `input_text_data.txt`. Файл має вміст, що показаний на рис. 8.1. Даний файл розміщений в тому ж каталозі на диску, що і файл проекту `Code::Blocks`, в іншому випадку — доведеться прописувати в програмі шлях до файлу.

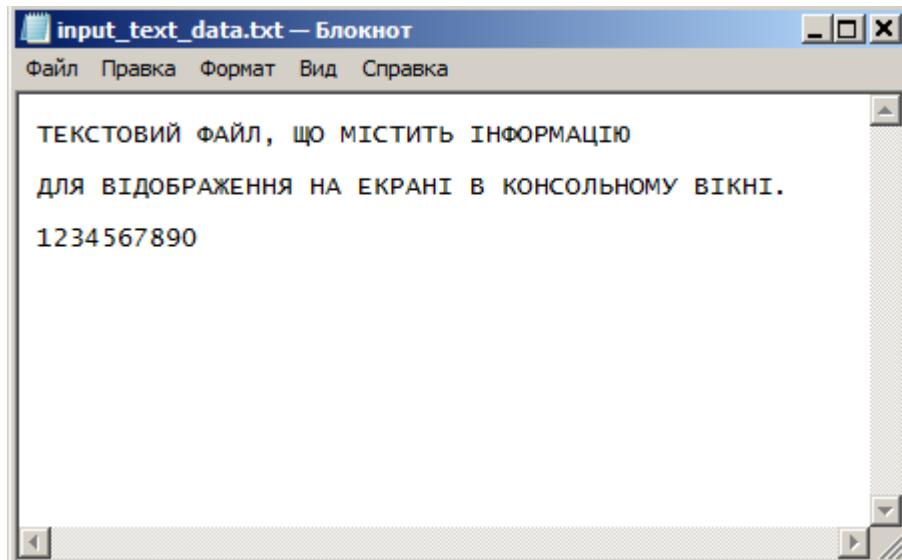


Рис. 8.1 — Приклад вмісту текстового файлу

Для коректного відображення текстової інформації в консольному вікні літера українського алфавіту «І» була замінена на літеру англійського алфавіту «I».

У випадку, якщо в програмі файл не вдалося відкрити, передбачено виконання відповідної перевірки і виведення повідомлення на екран.

За допомогою наступного рядка програми зчитується перший символ з файлу і присвоюється змінній `singleLetter`:

```
singleLetter = getc( pTestInputFile );
```

В циклі `while` виконується перевірка досягнення кінця файлу. Для цього використовується функція `feof()`. Поки в процесі зчитування даних з файлу символ кінця файлу не зустрівся — функція `feof()` повертає значення 0. В циклі `while` використовується операція логічне «НЕ», яка застосовується до результату, що повертається функцією `feof()`, таким чином, поки не досягнуто кінця файлу, тобто поки функція повертає значення 0, вираз-умова в циклі `while` буде `!0`, а враховуючи, що 0 — це логічна «хиба», то `!0` — це логічна «істина». Поки вираз-умова в циклі `while` істинна — відбувається виконання тіла циклу. В тілі циклу за допомогою функції `putc()` відбувається виведення на екран символу, який був зчитаний з файлу. Враховуючи, що функція `putc()` працює з

потоком, то вказавши в якості другого параметра — стандартний потік виведення `stdout` — дані виводяться на екран в консольне вікно:

```
putc( singleLetter, stdout );
```

Після цього за допомогою наступного оператора виконується зчитування наступного символу. Виконання циклу продовжується, поки не буде досягнуто кінця файлу. Результат виконання програми показано на рис. 8.2.

```
ТЕКСТОВИЙ ФАЙЛ, що містить інформацію  
для відображення на екрані в консольному вікні.  
1234567890  
Process returned 0 <0x0> execution time : 0.019 s  
Press any key to continue.
```

Рис. 8.2 — Виведення вмісту текстового файлу на екран в консольне вікно.

Вираз-умову в операторі циклу `while` можна було б записати інакше, записавши перевірку значення, яке зчитується із файлу і зберігається в змінній `singleLetter`. Поки ця змінна не дорівнює EOF — продовжувати виконання оператора циклу `while`. Нижче показаний фрагмент коду, який ілюструє зазначені зміни:

```
singleLetter = getc( pTestInputFile );  
while ( singleLetter != EOF ) {  
    putc( singleLetter, stdout );  
    singleLetter = getc( pTestInputFile );  
}
```

Нижче приводиться код програми, яка дозволяє виконувати зчитування даних з файлу, починаючи з кінця файлу в напрямку його початку. Для того, щоб перейти одразу в кінець файла використовується функція `fseek()`, а щоб дізнатися позицію кінця файла (тобто кількість символів, які записані між

початком файлу та символом кінця файлу) використовується функція `fseek()`, а результат зберігається в змінній `endPositionInFile`. Після цього в операторі циклу `while` на кожній ітерації відбувається збільшення величини зміщення відносно кінця файлу, що дозволяє на кожній ітерації переходити до зчитування нового символу в напрямку початку файлу, і виводити цей символ на екран. Змінна для зберігання зміщення називається `offsetRelativeEndPosInFile`. Таким способом вміст файла виводиться з кінця в напрямку початку.

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<locale.h>

int main()
{
    FILE * pTestInputFile;

    int singleLetter;

    int endPositionInFile;

    int offsetRelativeEndPosInFile;

    setlocale( LC_CTYPE, "Ukrainian" );

    pTestInputFile = fopen( "input_text_data.txt", "rb" );

    if( pTestInputFile == NULL ) {
        printf("\nПомилка при спробі відкрити файл!!!!");
        printf("\nНатисніть будь-яку клавішу для завершення");
        getch();
        exit( EXIT_FAILURE );
    }

    fseek( pTestInputFile, 0, SEEK_END );
    endPositionInFile = ftell( pTestInputFile );

    offsetRelativeEndPosInFile = 1;
```

```

while( offsetRelativeEndPosInFile <= endPositionInFile ){

    fseek( pTestInputFile, -offsetRelativeEndPosInFile, SEEK_END );

    singleLetter = getc( pTestInputFile );

    putc( singleLetter, stdout );

    ++offsetRelativeEndPosInFile;
}

putc( '\n', stdout );

fclose( pTestInputFile );

return 0;
}

```

На рис. 8.3 показано результат виконання програми по виведенню вмісту з файлу на екран, але виведення даних починається з кінця файла.

Рис. 8.3 — Результат виведення вмісту текстового файла з кінця файла в напрямку початку файла

Нижче приводиться приклад використання функцій `fwrite()` та `fread()` при записі та зчитуванні даних при роботі із двійковим (бінарним) файлом, а також використання функцій `fprintf()`, `fscanf()` при обміні даними із текстовим файлом. Коментарі щодо призначення відповідних змінних, а також призначення відповідних етапів обробки даних надано в коментарях програми. Аналізуючи текст програми можна помітити, що окремі частини програмного коду, які призначені для відкриття (або створення) файла і перевірки успішності виконання даних процедур, а також фрагмент коду по виділенню пам'яті для

динамічного масиву — повторюються в програмі. Тому, крім демонстрації прикладів окремих функцій по роботі із файлами, даний приклад демонструє певний недолік в організації написання програми, зокрема для поліпшення коду програми відповідні зазначені процедури можна реалізувати у вигляді окремих функцій, і там де потрібно — викликати лише одну відповідну функцію, замість того, щоб повторювати цілий фрагмент коду.

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<locale.h>

int main()
{
    FILE * pTestBinFile; // Вказівник для роботи з бінарним файлом
    FILE * pTestTextFile; // Вказівник для роботи з текстовим файлом

    int sizeOfDynArray; // Розмір одновимірного динамічного масиву

    int index; // Для зберігання індексу поточного елементу
               // при обробці масиву

    float * fltDynArray; // Вказівник на float виступає в якості
                         // імені динамічного масиву

    float temp; // При введенні значень елементів масиву
                 // з клавіатури спочатку введене значення
                 // зберігається в цій змінній, а потім
                 // присвоюється поточному елементу масиву

    setlocale(LC_CTYPE, "Ukrainian");
    printf("\nВведіть розмір масиву для збереження даних типу float.");
    printf("\nЯкщо задати розмір МЕНШЕ 1 – програма завершиться ");
    printf("\n\nРозмір: ");

    scanf("%d", &sizeOfDynArray);

    if( sizeOfDynArray < 1 )
        exit(EXIT_SUCCESS);
```

```

// Виділення пам'яті для одновимірного динамічного масиву
fltDynArray = (float *)malloc(sizeof(DynArray) * sizeof(float));
if( fltDynArray == NULL ) {
    printf("\n\nПам'ять для масиву не виділена ");
    printf("\nНатисніть клавішу для завершення програми ");
    getch();
    exit(EXIT_SUCCESS);
}

printf("\n\nВедіть значення елементів масиву:\n");
for( index = 0; index < sizeOfDynArray; index++ ) {
    printf("[%d] = ", index);
    scanf("%f", &temp);
    fltDynArray[index] = temp;
}

// Відкриття двійкового файлу в режимі запису
// Якщо файл відсутній, то він створиться

pTestBinFile = fopen( "test_bin_file.dat", "wb" );
if( pTestBinFile == NULL ) {
    printf("\nПомилка при спробі відкрити файл!!!\n");
    printf("\nНатисніть будь-яку клавішу для завершення\n");
    getch();
    exit(EXIT_FAILURE);
}

// запис у двійковий файл розміру масиву
fwrite( &sizeOfDynArray, sizeof(int), 1, pTestBinFile );

// запис у двійковий файл всього масиву
fwrite( &fltDynArray[0], sizeof(float), sizeOfDynArray, pTestBinFile );

// закриття двійкового файлу
fclose( pTestBinFile );

// обнулення елементів масиву та змінної, що зберігає розмір
sizeOfDynArray = 0;
for( index = 0; index < sizeOfDynArray; index++ )
    fltDynArray[index] = 0;

// звільнення пам'яті, яка була виділена для динамічного масиву
free( fltDynArray );

```

```

printf("\n\n Запис значень в двійковий файл виконано. ");
printf("\n Двійковий файл закрито.");
printf("\n Пам'ять для динамічного масиву - звільнено.");

printf("\n\n Перевірка правильності збереження даних");
printf("\n у двійковому файлі.");
printf("\n Відкриття двійкового файлу та зчитування з нього ");
printf("\n значення розміру масиву, а також значення ");
printf("\n елементів масиву.");
printf("\n Прочитані значення виводяться на екран та ");
printf("\n записуються в текстовий файл.");
}

// Перевірка правильності запису даних у двійковий файл.
// Двійковий файл відкривається в режимі читання.
// З двійкового файлу зчитується значення розміру масиву
// і присвоюється змінній, яка зберігає розмір.
// Виділяється пам'ять для динамічного масиву і
// зчитуються значення елементів масиву з файлу.
// Після цього значення розміру масиву, а також значення
// елементів масиву виводяться в консольне вікно, а також
// записуються в текстовий файл.

// Відкриття двійкового файлу в режимі читання

pTestBinFile = fopen( "test_bin_file.dat", "rb" );
if( pTestBinFile == NULL ) {
    printf("\nПомилка при спробі відкрити файл!!!\n");
    printf("\nНатисніть будь-яку клавішу для завершення\n");
    getch();
    exit(EXIT_FAILURE);
}

// зчитування значення змінної з двійкового файлу
fread( &sizeOfDynArray, sizeof(int), 1, pTestBinFile );

// Виділення пам'яті для одновимірного динамічного масиву
fltDynArray = (float *)malloc(sizeOfDynArray * sizeof(float));
if( fltDynArray == NULL ){
    printf("\n\nПам'ять для масиву не виділена\n");
    printf("\nНатисніть клавішу для завершення програми\n");
    getch();
    exit(EXIT_SUCCESS);
}

```

```

// зчитування елементів масиву з двійкового файлу
// fltDynArray == &fltDynArray[0].
// Оскільки ім'я масиву – це вказівник,
// значення якого відповідає адресі першого елементу масиву,
// можна записати ім'я динамічного масиву, а можна
// в явному вигляді записати адресу першого елементу масиву,
// застосовуючи операцію & до елемента з індексом 0.
fread( fltDynArray , sizeof(float) , sizeOfDynArray , pTestBinFile ) ;

// Відкриття текстового файла в режимі запису
// Якщо файл відсутній, то він створиться

pTestTextFile = fopen( "test_Text_file.txt" , "w" );
if( pTestTextFile == NULL ) {
    printf( "\nПомилка при спробі відкрити файл!!!\n" );
    printf( "\nНатисніть будь-яку клавішу для завершення\n" );
    getch();
    exit(EXIT_FAILURE);
}

// виведення повідомлення на екран
printf( "\n\nРозмір масиву: %d\n" , sizeOfDynArray );
printf( "\n\nЗначення елементів масиву: \n" );

// запис повідомлення в текстовий файл
fprintf( pTestTextFile , "\n\nРозмір масиву: %d\n" , sizeOfDynArray );

fprintf( pTestTextFile , "\n\nЗначення елементів масиву: \n" );

// виведення значень елементів масиву на екран в консольне вікно
// та в текстовий файл
for( index = 0; index < sizeOfDynArray; index++ ) {
    printf( "[%d]= %.2f\n" , index , fltDynArray[index] );
    fprintf( pTestTextFile , "[%d]= %.2f\n" , index , fltDynArray[index] );
}

// закриття файлів
fclose( pTestBinFile );
fclose( pTestTextFile );

// звільнення пам'яті, яка була виділена для динамічного масиву
free( fltDynArray );

return 0;
}

```

На рис. 8.4 приведено виконання програми для заданого з клавіатури значення розміру масиву та значень елементів масиву.

```
Введіть розмір масиву для збереження даних типу float.  
Якщо задати розмір МЕНШЕ 1 – програма завершиться  
  
Розмір: 4  
  
Ведіть значення елементів масиву:  
[0] = -0.5  
[1] = 2.125  
[2] = -9.75  
[3] = 105.0  
  
Запис значень в двійковий файл виконано.  
Двійковий файл закрито.  
Пам'ять для динамічного масиву – звільнено.  
  
Перевірка правильності збереження даних  
у двійковому файлі.  
Відкриття двійкового файла та зчитування з нього  
значення розміру масиву, а також значення  
елементів масиву.  
Прочитані значення виводяться на екран та  
записуються в текстовий файл.  
  
Розмір масиву: 4  
  
Значення елементів масиву:  
[0]= -0.50  
[1]= 2.13  
[2]= -9.75  
[3]= 105.00
```

Рис. 8.4 — Результат виконання програми

На рис. 8.5 показано вміст текстового файлу, що був створений в ході виконання програми, в який були записані значення розміру масиву, а також значення елементів масиву.

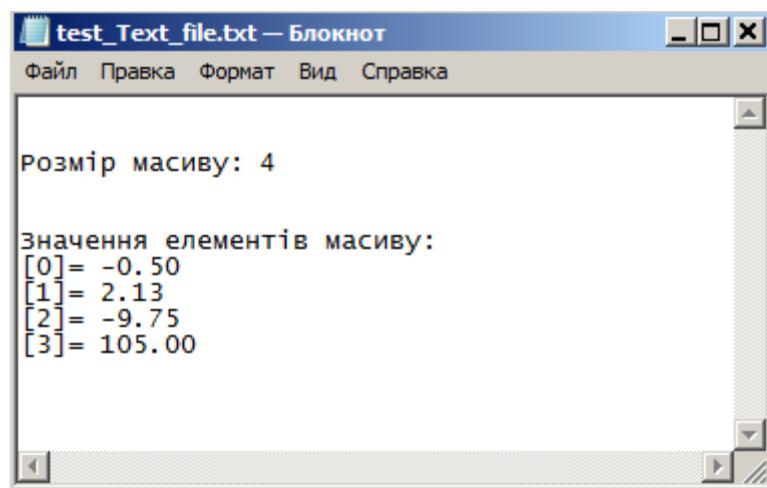


Рис. 8.5 — Вміст створеного текстового файлу із записаними в ньому даними

На рис. 8.6 показаний скріншот вмісту каталогу, в якому зберігаються файли проекту, а також створені два файли — двійковий файл, що має ім'я `test_bin_file.dat`, та текстовий файл, що має ім'я `test_Text_file.txt`.

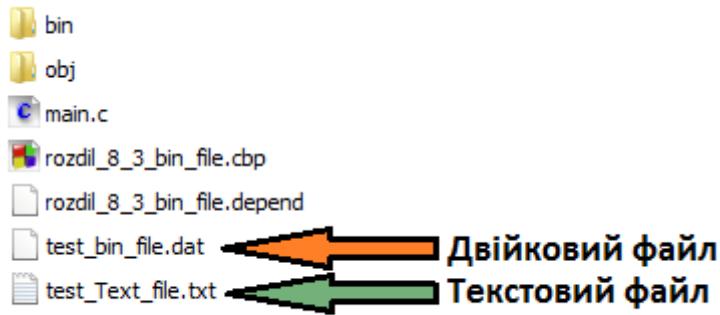


Рис. 8.6 — Вміст каталогу проекту, в якому розміщаються створені в програмі двійковий та текстовий файли

8.2. Функції по роботі із символічними рядками

При роботі із даними може виникати необхідність виконувати обробку текстової інформації, яка представлена у вигляді символічних рядків. Відмінність між символічним рядком і масивом символів полягає в тому, що в символічному рядку обов'язково повинен міститися символ кінця символічного рядка, — цей символ позначається `\0`. Це важлива відмінність між масивом символів (який може містити символи — букви, розділові знаки, цифри, але не містить символ `\0`) і символічним рядком [3, 4, 7]. Цілий ряд функцій, які орієнтовані на обробку символічних рядків, використовують символ `\0`, який позначає кінець символічного рядку, в своїй роботі. Часто, при роботі із символічними рядками або масивами символів, використовується тип даних `char`. Враховуючи, що символічний рядок обов'язково має містити символ `\0`, то при обранні розміру для відповідного масиву, який призначений для зберігання символічного рядку, необхідно передбачати розмір таким, щоб крім корисної інформації, залишався щонайменше один елемент масиву, який би зберігав символ `\0`. При використанні функцій `scanf()` або `fscanf()`, а також `printf()` або

`fprintf()` потрібно при роботі із символьними рядками використовувати специфікатор формату `%s`. Особливість функцій `scanf()` або `fscanf()` така, що вони читують символний рядок до першого пробільного символу. Тобто, якщо буде вводитися декілька слів, які розділені пробілами, то збережеться тільки перше слово. Для того, щоб вказати яку кількість символів функція `scanf()` або `fscanf()` повинна зчитати і зберегти в символний рядок, можна в специфікаторі формату вказати відповідну величину, як це показано в прикладі нижче:

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

#define STR_LENGTH 10

int main()
{
    char inputCharStr[STR_LENGTH] = { 0 };

    scanf("%9s", inputCharStr);

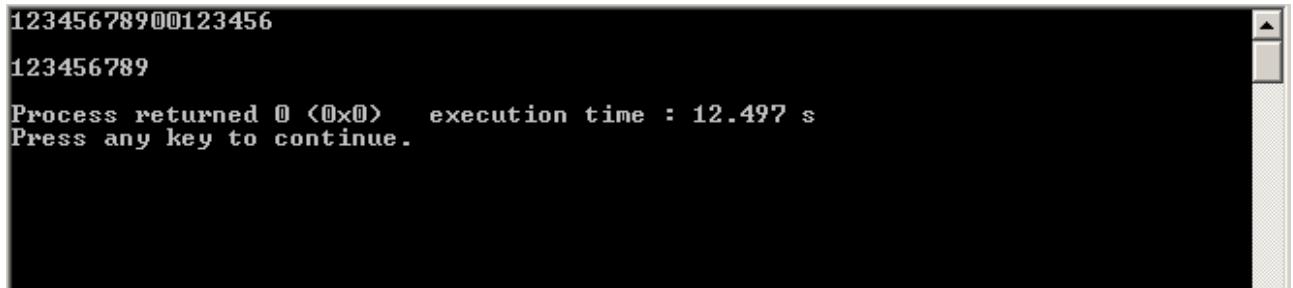
    printf("\n");
    printf("%s", inputCharStr);

    printf("\n");
    return 0;
}
```

В наведеному прикладі константа `STR_LENGTH` містить значення 10, яке виступає в якості розміру масиву `inputCharStr`. Оскільки даний масив призначений для зберігання символного рядка, тому має бути передбачене місце для символу кінця символного рядка — `\0`. Отже, масив повинен зберігати максимум 9 символів, які вводяться з клавіатури, а останній елемент масиву призначений для символу кінця рядка (рис. 8.7).

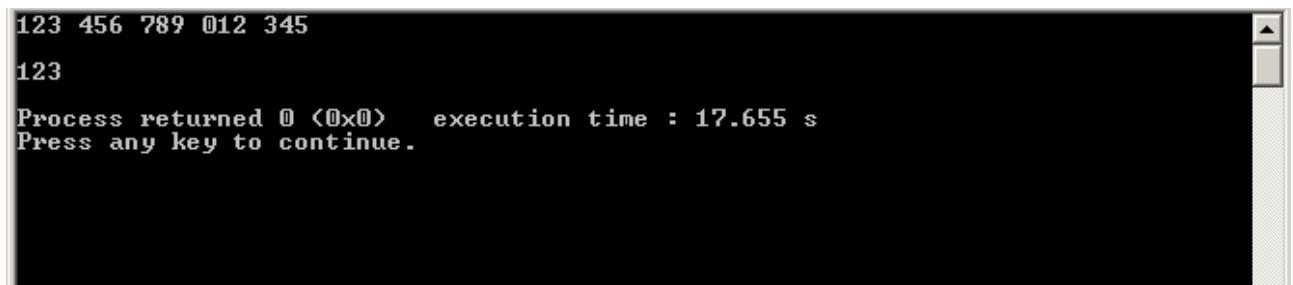
На рис. 8.8 показані символи, які були збережені в символному рядку, при введенні даних, що розділені пробілами. На рис. 8.8 видно, що в даному випадку, було збережене лише послідовність символів 123. Таке поводження

функції `scanf()` при використанні специфікатора формату `%s` при введені символьного рядка потрібно брати до уваги.



```
12345678900123456
123456789
Process returned 0 <0x0>    execution time : 12.497 s
Press any key to continue.
```

Рис. 8.7 — Збереження символьного рядка, який був введений з клавіатури



```
123 456 789 012 345
123
Process returned 0 <0x0>    execution time : 17.655 s
Press any key to continue.
```

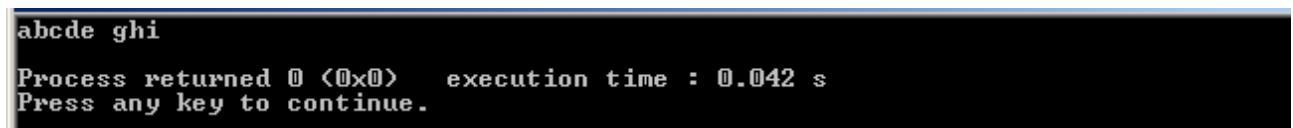
Рис. 8.8 — Збереження символьного рядка, при введенні послідовності символів, які розділені пробільним символом

Символьний рядок в програмі можна створити наступними способами:

1. Символьний рядок записується як масив символів, при цьому записується також символ `\0` як признак кінця рядка. Результат виведення рядка на рис. 8.9:

```
char inputCharStr[10] = { 'a', 'b', 'c', 'd', 'e', ' ', 'g', 'h', 'i', '\0' };

printf("%s", inputCharStr);
```



```
abcde ghi
Process returned 0 <0x0>    execution time : 0.042 s
Press any key to continue.
```

Рис. 8.9 — Виведення символьного рядка

2. Символьний рядок задається з урахуванням вибору такого розміру для рядка, щоб також міг бути збережений символ `\0`:

```
char inputCharStr[10] = "test text";
```

3. При записі символного рядка можна залишити квадратні дужки пустими, розмір необхідної пам'яті для збереження рядка буде визначений на основі довжини символного рядка:

```
char inputCharStr[ ] = "Input Data Requirements: Only Integers";
```

4. При оголошенні символного рядка можна використовувати вказівник, який призначений зберігати адресу блоку пам'яті за якою зберігається символний рядок, на рис. 8.10 показано виведення символного рядку в консольне вікно:

```
char * ptrInputCharStr = "Input Data Requirements: Only Integers";
printf("%s", ptrInputCharStr );
```

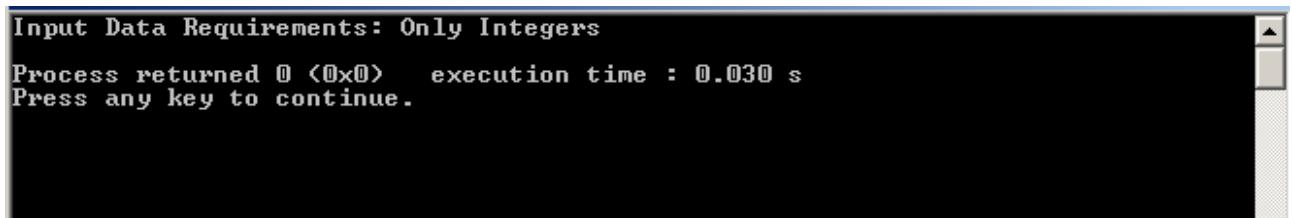


Рис. 8.10 — Виведення символного рядка в консольне вікно

Для введення символного рядка з клавіатури може використовуватися функція `gets()`, а для виведення символного рядка в консольне вікно можна застосовувати функцію `puts()`. Приклад застосування даних функцій показано нижче:

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

#define STR_LENGTH 100

int main()
{
    char inputCharStr[ STR_LENGTH ];

    puts("Enter input information less than 100 characters:");

    gets( inputCharStr );

    return 0;
}
```

Функція виконує зчитування вхідних символів, поки користувач не введе символ нового рядка (цей символ позначається \n), який генерується натисканням клавіші ENTER на клавіатурі. Функція gets() читає символ нового рядка і відкидає його, натомість записує символ \0, який позначає кінець рядка.

Функція gets() має наступний прототип [5]:

```
char * gets(char * Buffer);
```

В якості параметра функція отримує вказівник на блок пам'яті, куди необхідно зберігати символьний рядок. Функція повертає результат — вказівник на char. Якщо функція відпрацювала без помилок і успішно зчитала символьний рядок, вона повертає адресу блоку пам'яті, де зберігається введений символьний рядок. Якщо при роботі функції виникли помилки — вона повертає значення NULL. Значення NULL повертається також, якщо функція зустрічає символ кінця файлу. Враховуючи, що функція не виконує контроль кількості введених символів на етапі зчитування, це може стати причиною виходу за межі масиву при введенні символьного рядка з клавіатури.

Замість функції gets() рекомендується використовувати функцію fgets(), яка містить параметр, що визначає максимальну кількість символів (з урахуванням символу кінця рядка \0), які можуть бути збереженні. Для того, щоб мати можливість використовувати дану функцію при введенні даних з клавіатури, а не з файлу, на місці відповідного параметра замість вказівника на структуру FILE, потрібно вказати стандартний потік введення stdin. Приклад використання функції fgets() показано нижче.

```
#include <stdio.h>
#include <stdlib.h>
#define STR_LENGTH 100

int main()
{
    char inputCharStr[ STR_LENGTH ];
    puts("Enter input information less than 100 characters:");
    fgets( inputCharStr, STR_LENGTH, stdin );
    return 0;
}
```

Функція `puts()` використовується для виведення символального рядка в консольне вікно, має такий прототип [5, 7]:

```
int puts(const char * Str);
```

Функція в якості параметра отримує вхідний рядок і виводить його в консольне вікно. Якщо функція `puts()` відпрацювала без помилок, вона повертає невід'ємне ціле значення. Якщо в процесі роботи функції `puts()` виникли помилки — функція повертає значення EOF. При виведенні символального рядка на екран в консольне вікно, функція додає в кінці рядка символ нового рядка `\n`, тим самим переводячи курсор на новий рядок.

Для виведення символального рядка також може застосовуватися функція `fputs()`. При виклику функції на позиції первого параметру записується символний рядок, який виводиться, а на позиції другого параметра — стандартний потік виведення `stdout`. На відміну від функції `puts()`, функція `fputs()` не додає символ нового рядка `\n` при виведенні символального рядка.

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

#define STR_LENGTH 100

int main()
{
    char inputCharStr[ STR_LENGTH ];

    fputs( "Enter input information less than 100 characters: ", stdout );
    fgets( inputCharStr, STR_LENGTH, stdin );

    return 0;
}
```

Для роботи із символними рядками передбачений цілий ряд функцій, щоб мати можливість їх використовувати необхідно підключити заголовочний файл `string.h`.

Наприклад, функція `strlen()` використовується для визначення довжини символального рядка. Функція має такий прототип [5, 7, 8]:

```
size_t strlen(const char * Str);
```

Функція отримує в якості параметра символічний рядок і повертає довжину символічного рядка, без врахування символу кінця рядка. Наприклад, якщо заданий символічний рядок:

```
char * str = "123";
```

то для наступного прикладу виклику функції `strlen()`, дана функція поверне значення 3:

```
strlen(str);
```

Функція `strcpy()` дозволяє виконувати копіювання символічних рядків. Прототип цієї функції такий [5, 8]:

```
char * strcpy(char * strDestination, const char * strSource);
```

Функція отримує два параметри, перший параметр — це символічний рядок, що буде отримувати копію символічного рядка, який стоїть на місці другого параметра. При копіюванні символічного рядка необхідно щоб символічний рядок `strDestination` (який отримує копію) мав відповідний розмір.

Функція `strcat()` дозволяє приєднувати вміст одного символічного рядка до іншого. Прототип функції `strcat()` такий [5, 8]:

```
char * strcat(char * strDestination, const char * strSource);
```

За допомогою даної функції копія символічного рядка `strSource` (який стоїть на місці другого параметра в переліку параметрів функції `strcat()`) записується в кінець символічного рядка `strDestination`. Символ кінця рядку в символічному рядку `strDestination` перезаписується першим символом символічного рядка `strSource`. Функція не виконує перевірку чи достатній розмір символічного рядка `strDestination`, щоб зберігати результат об'єднання символічних рядків.

Функція `strchr()` використовується для пошуку заданого символу в символічному рядку. Функція `strchr()` має наступний прототип [5]:

```
char * strchr(const char * Str, int Value);
```

Перший параметр — це символічний рядок, в якому відбувається пошук символу, що записується на місці другого параметру `Value`. Якщо функція не знаходить шуканий символ — вона повертає значення `NULL`. Якщо функція знаходить символ — вона повертає вказівник на перший знайдений символ в символному рядку `Str`, який дорівнює шуканому.

Ознайомитися із додатковою довідковою інформацією щодо функцій, які дозволяють виконувати обробку символічних рядків, можна в [4—8].

8.3. Питання для самоконтролю

1. Поясніть яка відмінність в особливостях збереження даних при їх записі в текстовий файл та в двійковий (бінарний) файл.
2. Чим відрізняється режим відкриття файлу `w` та `w+`?
3. Яке значення буде повертати функція `fopen()` в разі відсутності файлу, який відкривається в програмі в режимі `r`?
4. Що буде відбуватися, якщо в програмі за допомогою функції `fopen()` відкривається файл в режимі `w`, але при цьому файл із вказаним ім'ям на диску відсутній?
5. Яка різниця між масивом символів та символічним рядком?
6. Напишіть програму, яка підраховує кількість символів, які вводяться з клавіатури.
7. Яку небезпеку становить використання функції `gets()` при зчитуванні символічного рядку?
8. Яка перевага функції `fget()` порівняно із функцією `gets()`?
9. Якщо змінна типу `char`, що має значення 125, дозаписується в кінець у текстовий файл та в бінарний файл, то на скільки байтів зміниться розмір кожного із зазначених файлів? На скільки байтів зміниться розмір кожного файла, якщо вказана змінна буде мати значення 99?

9. СТРУКТУРИ. ОСНОВИ РОБОТИ ІЗ СТРУКТУРАМИ

При вирішенні багатьох практичних задач може бути недостатнім використання окремо лише змінних, констант, масивів чи символічних рядків, і для представлення даних, над якими повинна проводитись обробка, доречно застосовувати структури [3—7].

Структура представляє собою тип даних, який характеризується поєднанням під одним іменем елементів, що можуть мати різний тип даних, і певним чином логічно бути пов'язаними між собою. Можливість використовувати в програмі структурний тип даних (структурну) дозволяє більш цілісно і більш природно виконувати опис тих об'єктів над якими необхідно проводити відповідну обробку. При роботі із структурою необхідно виконати її оголошення (або іншими словами, задати шаблон структури), що в загальному випадку записується в програмі наступним чином [5, 7]:

```
struct ім'я_структурі {  
    тип_даних ім'я_елементу_структурі;  
    тип_даних ім'я_елементу_структурі;  
    ...  
    тип_даних ім'я_елементу_структурі;  
};
```

Ім'я структури (або ще по-іншому називають — *тег структури*) — це будь-який правильно складений ідентифікатор. Аналогічно до вибору імен для змінних, констант, функцій, масивів тощо, при виборі імені структурного типу необхідно обирати осмисленні імена, які в своїй назві відображають призначення створеного структурного типу даних. Елементи структури (або по-іншому їх ще називають *поля структури*) можуть бути змінними різних типів, масивами, символічними рядками, вказівниками тощо. При записі шаблону структури надається інформація щодо схеми представлення даних — типів даних полів (елементів) структури і послідовність їх слідування в шаблоні структури [3—7].

9.1. Оголошення змінних структурного типу даних

Оголошення в програмі змінної структурного типу відбувається схожим чином як це виконувалось для оголошення змінних базових типів даних (zmінних типу int, float або double та ін.), тільки в цьому випадку необхідно записувати службове слово `struct` та ім'я структури (тег структури), що вказувалося при попередньому створенні шаблону структури. Схематично, оголошення змінної структурного типу можна записати таким чином [5, 7]:

```
struct ім'я_структурі ім'я_zmінної_структурного_типу;
```

Структурний тип даних (або коротко — структура) може бути корисним коли необхідно оперувати даними для опису яких базові типи даних є недостатніми. Наприклад, для запису комплексного числа, яке складається з дійсної та уявної частини, відсутній відповідний тип даних, тому розробник за допомогою структури може створити необхідний йому тип даних і використовувати його для рішення поставлених задач.

Для того щоб звернутися до відповідного поля структури (елементу структури) з метою задати значення даному полю, або використати його значення для певних задач, необхідно вказати ім'я змінної структурного типу, поставити символ «крапка», а потім вказати ім'я відповідного поля структури. Наприклад, присвоєння деякого значення певному полю змінної структурного типу можна виконати таким чином [5, 7]:

```
ім'я_zmінної_structурного_типу.ім'я_елементу_структурі = значення;
```

При зверненні до елементів (до полів) змінної структурного типу потрібно враховувати їх типи даних, які були зазначені при записі шаблону структури.

Змінну структурного типу можна ініціалізувати при її оголошенні, задавши значення відповідних полів.

Нижче приводиться лістинг програми, що ілюструє принципи роботи із змінними структурного типу на прикладі структури `complexNumber`, яка створюється в програмі. Данна структура призначена для роботи із комплексними числами. Структура `complexNumber` складається із двох елементів (двох полів), які мають назву `real` та `imaginary`, і призначенні для зберігання дійсної та уявної частини комплексного числа, обидва елементи структури мають тип даних `double`:

```
struct complexNumber {  
    double real;  
    double imaginary;  
};
```

В програмі ініціалізується змінна на ім'я `x` структурного типу. При ініціалізації змінної структурного типу потрібно у фігурних дужках записати значення відповідних елементів (полів). Крім того, в програмі оголошуються змінні структурного типу з іменами `y` та `result`.

Значення структурної змінної `y` (значення полів `real` та `imaginary`) задаються користувачем з клавіатури. Введення з клавіатури значення для змінної на ім'я `y` відбувається із використанням функції `scanf()`. В даному випадку аналогічно введенню значень змінних базових типів даних, при введені значень для елементів (полів) `real` та `imaginary` необхідно використовувати операцію «отримати адресу», яка позначається символом амперсанд — `&`. Наприклад, введення значення дійсної частини комплексного числа реалізується в представленому прикладі наступним чином:

```
printf("Дійсна частина = ");  
scanf("%lf", &y.real);
```

Введення значення уявної частини комплексного числа виконується аналогічно:

```
printf("Уявна частина = ");  
scanf("%lf", &y.imaginary);
```

Зміна `result` містить результат додавання змінних `x` та `y` (при додаванні необхідно окремо знайти значення суми полів, які використовуються для зберігання дійсної частини, та значення суми полів, які використовуються для

зберігання уявної частини, і зберегти результат у відповідні елементи (поля) змінної `result`). Результат додавання виводиться на екран в консольному вікні. При виведенні значень відповідних полів змінної `result` потрібно обирати відповідні специфікатори формату для коректного відображення значень на екрані. Для виведення полів змінних, які зберігають дійсну та уявну частини числа у форматі запису $a \pm bj$ (для позначення уявної одиниці застосовується символ `j`), використовуються відповідні модифікатори та специфікатори формату функції `printf()`:

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

// шаблон структури
struct complexNumber {
    double real;
    double imaginary;
};

int main()
{
    // ініціалізація змінної структурного типу
    struct complexNumber x = { 1.5, -4.0 };

    // оголошення змінних структурного типу
    struct complexNumber y, result;

    setlocale(LC_CTYPE, "Ukrainian");
    printf("Введіть дійсну та уявну частини комплексного числа у:\n");

    printf("Дійсна частина = ");
    scanf("%lf", &y.real);

    printf("Уявна частина = ");
    scanf("%lf", &y.imaginary);

    // знаходження суми дійсних частин доданків
    result.real = x.real + y.real;

    // знаходження суми уявних частин доданків
    result.imaginary = x.imaginary + y.imaginary;

    // виведення результатів в консольне вікно
    printf("\n\nПерший доданок:\n");
    printf("x = %.2lf%+.2lfj", x.real, x.imaginary);

    printf("\n\nДругий доданок:\n");
    printf("y = %.2lf%+.2lfj", y.real, y.imaginary);

    printf("\n\nСума:\n");
    printf("result = %.2lf%+.2lfj \n\n", result.real, result.imaginary);
}
```

```
    return 0;  
}
```

На рис. 9.1 показано результат виконання програми.

The screenshot shows a terminal window with the following output:

```
Введіть дійсну та уявну частини комплексного числа у:  
Дійсна частина = -5.45  
Уявна частина = 0.28  
  
Перший доданок:  
x = 1.50-4.00j  
  
Другий доданок:  
y = -5.45+0.28j  
  
Сума:  
result = -3.95-3.72j  
  
Process returned 0 (0x0)   execution time : 18.409 s  
Press any key to continue.
```

Рис. 9.1 — Виведення в консольне вікно результату виконання програми, що знаходить суму двох комплексних чисел

Якщо виникає необхідність виконати ініціалізацію декількох змінних структурного типу, це можна зробити схожим чином як це виконувалося при роботі із змінними базових типів даних, але при цьому потрібно у фігурних дужках вказувати значення відповідних полів. Наприклад, в попередній програмі можна замість введення значення змінної *у* з клавіатури, задати її значення безпосередньо в програмі:

```
struct complexNumber x = { 1.5, -4.0 }, y = { 3.5, -1.4 };
```

При вирішенні багатьох задач часто виникає потреба опрацьовувати масиви даних, причому такі масиви можуть зберігати значення структурного типу. В такому випадку доводиться мати справу із масивами структур.

9.2. Масиви структур та вказівники на структури

В разі необхідності можна створювати масиви структурного типу. Нехай в попередній програмі замість окремих змінних використовується масив, що має ім'я *X*. Розмір даного масиву задається за допомогою директиви препроцесора `#define`. Значення елементів масиву структурного типу

ініціалізуються в програмі шляхом запису значень відповідних полів для кожного елементу масиву. В програмі виконується знаходження елементу масиву, в якому зберігається комплексне число із найбільшим модулем (якщо комплексне число позначити $c=a+jb$, то модуль комплексного числа буде визначатися наступним чином: $\sqrt{a^2 + b^2}$). В ході виконання програми на екран монітора в консольне вікно повинно виводитись значення індексу елемента масиву, що містить найбільший модуль серед тих комплексних чисел, які зберігаються в масиві, а також виводиться безпосередньо саме комплексне число і обраховане значення модуля:

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <math.h>

#define N 5

// шаблон структури
struct complexNumber {
    double real;
    double imaginary;
};

int main()
{
    // Ініціалізація елементів масиву X, які мають
    // структурний тип struct complexNumber

    struct complexNumber X[N] = {
        {1.0, -0.5}, // 1-й елемент масиву
        {2.0, 4.5}, // 2-й елемент масиву
        {4.2, -3.8}, // 3-й елемент масиву
        {-0.7, -1.9}, // 4-й елемент масиву
        {-0.4, 0.2}, // 5-й елемент масиву
    };

    // Змінна maxModul для зберігання максимального значення
    // модуля комплексного числа серед комплексних чисел, які
    // зберігаються в масиві X
    // Змінна temp використовується в процесі пошуку
    // комплексного числа із максимальним модулем

    double maxModul, temp;
```

```

// Змінна i використовується для зберігання поточного
// значення індексу елемента масиву
// Змінна m використовується для зберігання
// індексу елемента масиву, що має максимальний модуль

int i, m;

setlocale(LC_CTYPE, "Ukrainian");

// Виведення значень елементів масиву X на екран
// в форматі: [індекс_елементу]: a+bj,
// де a - дійсна частина комплексного числа,
// b - уявна частина комплексного числа.

printf("\nЗначення елементів масиву:\n");
for( i = 0; i < N; i++ )
    printf("[%d]: %.2lf%+.2lfj \n", i, X[i].real, X[i].imaginary );




// Пошук комплексного числа в масиві, що має максимальний модуль
i = 0;
m = i;
maxModul = sqrt( pow( X[i].real, 2 ) + pow( X[i].imaginary, 2 ) );

for( i = 1; i < N; i++ ){
    temp = sqrt( pow( X[i].real, 2 ) + pow( X[i].imaginary, 2 ) );

    if( maxModul < temp){
        m = i;
        maxModul = temp;
    }
}

printf("\n\n-----\n");

printf("\nЕлемент масиву, що має максимальний модуль\n");
printf("[%d]: %.2lf%+.2lfj \n", m, X[m].real, X[m].imaginary );
printf("Модуль: %.2lf", maxModul );

printf("\n\n");

return 0;
}

```

На рис. 9.2 показано вигляд консольного вікна із виведеним результатом виконання програми.

```
Значення елементів масиву:  
[0]: 1.00-0.50j  
[1]: 2.00+4.50j  
[2]: 4.20-3.80j  
[3]: -0.70-1.90j  
[4]: -0.40+0.20j  
  
-----  
Елемент масиву, що має максимальний модуль  
[2]: 4.20-3.80j  
Модуль: 5.66  
  
Process returned 0 (0x0) execution time : 0.025 s  
Press any key to continue.
```

Рис. 9.2 — Результат виконання програми

В ході виконання представленої програми виникає необхідність оперувати елементами масиву, що мають структурний тип даних `struct complexNumber`. Масив має ім'я `X`. Щоб звертатися до відповідного елемента масиву потрібно вказувати його індекс. Якщо змінна `i` використовується в якості індексу елемента масиву, то звернення до елементу масиву відбувається таким чином: `X[i]`. Але в даному разі робота ведеться із масивом, що має структурний тип, тому крім імені масиву та індексу елементу, необхідно також вказувати ім'я поля структури при реалізації відповідної обробки даних. Тому для представленого вище прикладу програми, в якій використовується масив структур, для доступу до значення відповідного поля структури відповідного елементу масиву використовується наступний формат запису:

`ім'я_масиву[індекс_елементу_масиву].ім'я_поля_структурні`

При роботі із структурами може виникати необхідність використовувати вказівники на структуру. Вказівник на структуру зберігає адресу першого байту області пам'яті, яка виділяється для зберігання, наприклад, змінної структурного типу або масиву структур. Розглянемо особливості роботи із вказівниками на структуру на прикладі. Нехай в програмі використовується структура, яка дозволяє організувати збереження та обробку комплексних чисел:

```
struct complexNumber {  
    double real;  
    double imaginary;  
};
```

Нехай в програмі ініціалізується змінна `t` структурного типу:

```
struct complexNumber t = {0.5, -2.5};
```

Змінна `t` складається із двох полів — `real` та `imaginary`, які мають тип даних `double`. Для типу даних `double` виділяється 8 байт пам'яті. Таким чином, змінна `t` структурного типу, яка містить два поля по 8 байт — займає в пам'яті 16 байт. Враховуючи, що кожний байт пам'яті має свою адресу — вказівник на структуру зберігає адресу першого байту області пам'яті, яка виділена для зберігання змінної `t`, і доступ до значень полів змінної `t` може бути організовано через використання вказівника на структуру, якому було присвоєно адресу змінної `t`.

Оголошення вказівника на структуру виконується схожим чином як це відбувалося при роботі із вказівниками базових типів даних. Нехай вказівник на структуру має ім'я `ptr` та оголошується наступним чином:

```
struct complexNumber *ptr;
```

Вказівнику присвоюється адреса змінної `t`:

```
ptr = &t;
```

Тепер, використовуючи вказівник `ptr`, можна звертатися до полів `real` та `imaginary` змінної `t` з метою, наприклад, доступу до їх значень, або з метою присвоєння їм деяких значень в ході виконання програми.

При зверненні до полів структури, використовуючи для цього вказівник на структуру, можна використовувати два різні способи.

Перший спосіб — це застосування операції «доступ за адресою», яка позначається символом `*`. В загальному випадку можна це записати таким чином [3, 7]:

(*вказівник_на_структур) .ім'я_поля_структур

При використанні такої форми запису потрібно бути уважним, оскільки операція розіменування (або іншими словами — «доступ за адресою»), яка позначається символом * має менший пріоритет ніж операція доступу до елементу (поля) структури, яка позначається символом «крапка». Враховуючи, що спочатку треба виконати розіменування вказівника, щоб мати можливість звернутися до поля структури, адреса в пам'яті якої зберігається у вказівнику, тому потрібно використати круглі дужки для того, щоб сформувати правильним чином потрібну послідовність виконання операцій.

Другий спосіб — це використання операції доступу до елементу структури, яка записується за допомогою двох знаків — «мінус» і «більше», і має такий вид: ->

При другому способі звернення до полів (елементів) структури, використовуючи при цьому вказівник на структуру, форма запису в загальному вигляді буде такою [3, 7]:

вказівник_на_структур->ім'я_поля_структур

В наведеному нижче прикладі демонструється використання вказівника на структуру, а також обидва способи доступу до елементів структури. Вказівник на структуру використовується щоб дістатися до значення поля структури, а також для зміни цього значення. Загалом, робота із вказівниками буде схожим чином як це було розглянуто в попередніх розділах на прикладах при роботі із змінними базових типів даних та при роботі із масивами і символічними рядками.

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

// Шаблон структури
struct complexNumber {
    double real;
    double imaginary;
};
```

```

int main()
{
    // Ініціалізація структурної змінної на ім'я t
    struct complexNumber t = {0.5, -2.5};

    // Вказівник на структуру
    struct complexNumber *ptr;

    // Присвоєння вказівнику ptr адреси змінної, що має ім'я t
    ptr = &t;

    // Виведення на екран комплексного числа у форматі
    // a+bj, при цьому для доступу до полів real та imaginary
    // структурної змінної з ім'ям t, адреса якої
    // зберігається у вказівнику ptr, використовуються
    // два різні способи запису

    setlocale(LC_CTYPE, "Ukrainian");

    printf("\nПочаткове значення змінної t:\n");
    printf("%.2lf%+.2lfj \n\n", ptr->real, (*ptr).imaginary);

    // Зміна значень полів структурної змінної t,
    // використовуючи для цього вказівник на структуру ptr.
    // Використовуються два способи доступу до полів структури

    (*ptr).real = -9.0;
    ptr->imaginary = 4.2;

    // Виведення на екран значення структурної змінної,
    // використовуючи ім'я змінної t.

    printf("Нове значення змінної t:\n");
    printf("%.2lf%+.2lfj \n\n", t.real, t.imaginary);

    return 0;
}

```

На рис. 9.3 показано результат виконання програми.

```
Початкове значення змінної t:  
0.50-2.50j  
  
Нове значення змінної t:  
-9.00+4.20j  
  
Process returned 0 (0x0) execution time : 0.025 s  
Press any key to continue.
```

Рис. 9.3 — Результати виконання програми, в якій відбувається зміна значення структурної змінної за допомогою використання вказівника на структуру

Вказівники часто доводиться застосовувати в задачах, в яких необхідно виконувати обробку масивів. Розглянемо приклад, в якому робота ведеться із масивом структур. Шаблон структури оберемо такий самий, що вже був раніше використаний, і передбачає роботу із комплексними числами. В програмі створюється масив структур. Передбачається, що робота виконується із динамічним масивом. Таким чином, для роботи із динамічним масивом необхідно скористатися вказівником. Оскільки в даному випадку, виконується обробка значень структурного типу, тому і вказівник, який буде виступати в якості імені динамічного масиву структур — повинен бути вказівником на відповідну структуру, шаблон якої записується в програмі. Кожен елемент масиву призначений для зберігання комплексного числа, тобто кожен елемент масиву — це структура, що містить два поля — `real` та `imaginary`. Значення цих полів для кожного елементу масиву можна задавати з клавіатури, або обирати варіант, при якому значення полів для кожного елементу масиву генеруються автоматично. Вибір робиться користувачем на етапі виконання програми. Необхідно знайти в масиві комплексне число, що має найбільший модуль. При цьому, на відміну від раніше розглянутого варіанту рішення цього завдання, коли пошук проводився в статичному масиві, для зберігання знайденого значення буде використовуватися вказівник на структуру (це буде інший вказівник, ніж той, який буде зберігати адресу першого байту виділеної області пам'яті, яка виділяється для зберігання елементів масиву структурного типу). В результаті виконання алгоритму пошуку — вказівник на структуру

буде містити адресу елемента, що має найбільший модуль. За допомогою цього вказівника значення елементу масиву буде виведено на екран в консольне вікно. В програмі застосовуються різні стилі звертання до елементів масиву, використовуючи для цього вказівник на структуру. В деяких записах в явному вигляді за допомогою круглих дужок було визначено послідовність виконання операцій при роботі із вказівниками на етапі введення даних з клавіатури, а також при використанні значень елементів масиву в ході виконання їх обробки. Надані коментарі в тексті програми пояснюють призначення відповідних змінних.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <math.h>
#include <conio.h>

// Шаблон структури
struct complexNumber {
    double real;
    double imaginary;
};

int main()
{
    // Вказівник з ім'ям pX буде зберігати адресу першого байту
    // пам'яті, яка буде виділятися для зберігання масиву структур
    // в ході виконання програми. Вказівник pX може розглядатися
    // як ім'я динамічного масиву, в якому кожен елемент масиву
    // представляє собою структуру struct complexNumber
    // Вказівник ptr буде зберігати адресу елементу масиву,
    // який має найбільший модуль.
    struct complexNumber *pX, *ptr;

    pX = NULL;
    ptr = NULL;

    // Змінна, що призначена для зберігання розміру масиву структур
    int sizeOfArray;
```

```

// Змінна, що призначена для зберігання значення щодо способу
// введення значень елементів масиву структур.
// Якщо змінна дорівнює 1 – значення масиву задаються з клавіатури
// Якщо змінна дорівнює 2 – елементи масиву заповнюються
// автоматично
int settingMode;

// Змінна використовується в якості індексу елементу масиву
int i;

// Змінна, що використовується для формування значення
// дійсної та уявної частини комплексного числа при виборі
// варіанту автоматичного заповнення масиву.
// Даній змінній присвоюється згенероване випадкове значення,
// після цього сформоване значення присвоюється відповідному
// полю поточного елемента масиву
double randNum;

// Змінна використовується для формування знаку числа
// дійсної та уявної частини елементу масиву, при
// виборі автоматичного заповнення масиву
int signNum;

// Змінні, що використовуються при пошуку
// комплексного числа в масиві, що має найбільший модуль
double temp, maxMod;
int indexOfMaxMod;

// Введення з клавіатури значення розміру масиву
setlocale(LC_CTYPE, "Ukrainian");

printf("\nВведіть розмір масиву: ");
scanf("%d", &sizeOfArray);

while( sizeOfArray <= 0 ) {
    printf("\nРозмір масиву повинен бути більше 0");
    printf("\nВведіть розмір масиву: ");
    scanf("%d", &sizeOfArray );
}

```

```

pX = calloc( sizeOfArray, sizeof (struct complexNumber) ) ;

if( pX == NULL ) {
    printf("Увага! Пам'ять не була виділена");
    printf("\nНатисніть клавішу для завершення програми...") ;
    getch();
    exit(EXIT_FAILURE);
}

printf("\n\nОберіть спосіб задання значень елементам масиву:");
printf("\n1 - ввести з клавіатури");
printf("\n2 - сформувати значення автоматично");
printf("\nВведіть номер обраного способу: ");
scanf("%d", &settingMode);

while( settingMode != 1 && settingMode != 2 ) {
    printf("\nХибний вибір. Спробуйте ще раз.");
    printf("\nОберіть спосіб задання значень елементам масиву:");
    printf("\n1 - ввести з клавіатури");
    printf("\n2 - сформувати значення автоматично");
    printf("\nВведіть номер способу: ");
    scanf("%d", &settingMode);
}

switch( settingMode )
{
    case 1:
        printf("\n\nЗадайте значення комплексних чисел, ");
        printf("які зберігаються в масиві:\n");

        for( i = 0; i < sizeOfArray; i++ ) {
            printf("\nЕлемент масиву [%d]", i );
            printf("\nДійсна частина: ");
            scanf("%lf", &((pX+i)->real) );
            printf("Уявна частина: ");
            scanf("%lf", &((*(pX+i)).imaginary) );
            //scanf("%lf", &(pX[i].imaginary) );
        }

        printf("\n\nВи задали такі значення:");

        for( i = 0; i < sizeOfArray; i++ ) {
            printf("\n[%d]: ", i );
            printf("%.2lf%+.2lfj", pX[i].real, (pX+i)->imaginary );
        }

        break;
}

```

```

    case 2:
        printf("\n\nЗгенеровано такі значення:");

        for( i = 0; i < sizeOfArray; i++ ){

            randNum = rand()%10 + (rand()%101) / 100.0;
            signNum = rand()%2;
            pX[i].real = pow(-1, signNum) * randNum;

            randNum = rand()%10 + (rand()%101) / 100.0;
            signNum = rand()%2;
            (pX+i)->imaginary = pow(-1, signNum) * randNum;

            printf("\n[%d]: ", i );
            printf("%.2lf%+.2lfj", (* (pX+i)).real, pX[i].imaginary );
        }

        break;
    }

// Пошук елемента масиву, що має найбільший модуль
// Адреса знайденого елемента масиву присвоюється вказівнику ptr

// Спочатку вказівнику ptr присвоюється адреса першого елемента масиву
ptr = pX;
maxMod = sqrt( pow( (*ptr).real ,2) + pow( ptr->imaginary,2) );
indexOfMaxMod = 0;

for( i = 1; i < sizeOfArray; i++ ){
    temp = sqrt( pow(pX[i].real,2) + pow(pX[i].imaginary,2) );

    if( maxMod < temp ){
        ptr = pX + i;
        maxMod = temp;
        indexOfMaxMod = i;
    }
}

printf("\n\n-----\n");
printf("Комплексне число із найбільшим модулем: ");
printf("\n[%d]: ", indexOfMaxMod );
printf("%.2lf%+.2lfj", (*ptr).real, ptr->imaginary );
printf("\nМодуль: %.2lf", maxMod );

printf("\n\n");
return 0;
}

```

На рис. 9.4 приведено результати виконання програми по знаходженню в масиві комплексного числа, що має найбільший модуль. З клавіатури користувачем було задано розмір масиву, який був обраний рівним 5. А також користувачем було введено значення дійсної та уявної частини для кожного комплексного числа.

```
Введіть розмір масиву: 5

Оберіть спосіб задання значень елементам масиву:
1 - ввести з клавіатури
2 - сформувати значення автоматично
Введіть номер обраного способу: 1

Задайте значення комплексних чисел, які зберігаються в масиві:

Елемент масиву [0]
Дійсна частина: -2.5
Уявна частина: 0.8

Елемент масиву [1]
Дійсна частина: 3.2
Уявна частина: -0.7

Елемент масиву [2]
Дійсна частина: 5.0
Уявна частина: -3.5

Елемент масиву [3]
Дійсна частина: 0.9
Уявна частина: 6.1

Елемент масиву [4]
Дійсна частина: 2.4
Уявна частина: 1.8

Ви задали такі значення:
[0]: -2.50+0.80j
[1]: 3.20-0.70j
[2]: 5.00-3.50j
[3]: 0.90+6.10j
[4]: 2.40+1.80j

-----
Комплексне число із найбільшим модулем:
[3]: 0.90+6.10j
Модуль: 6.17

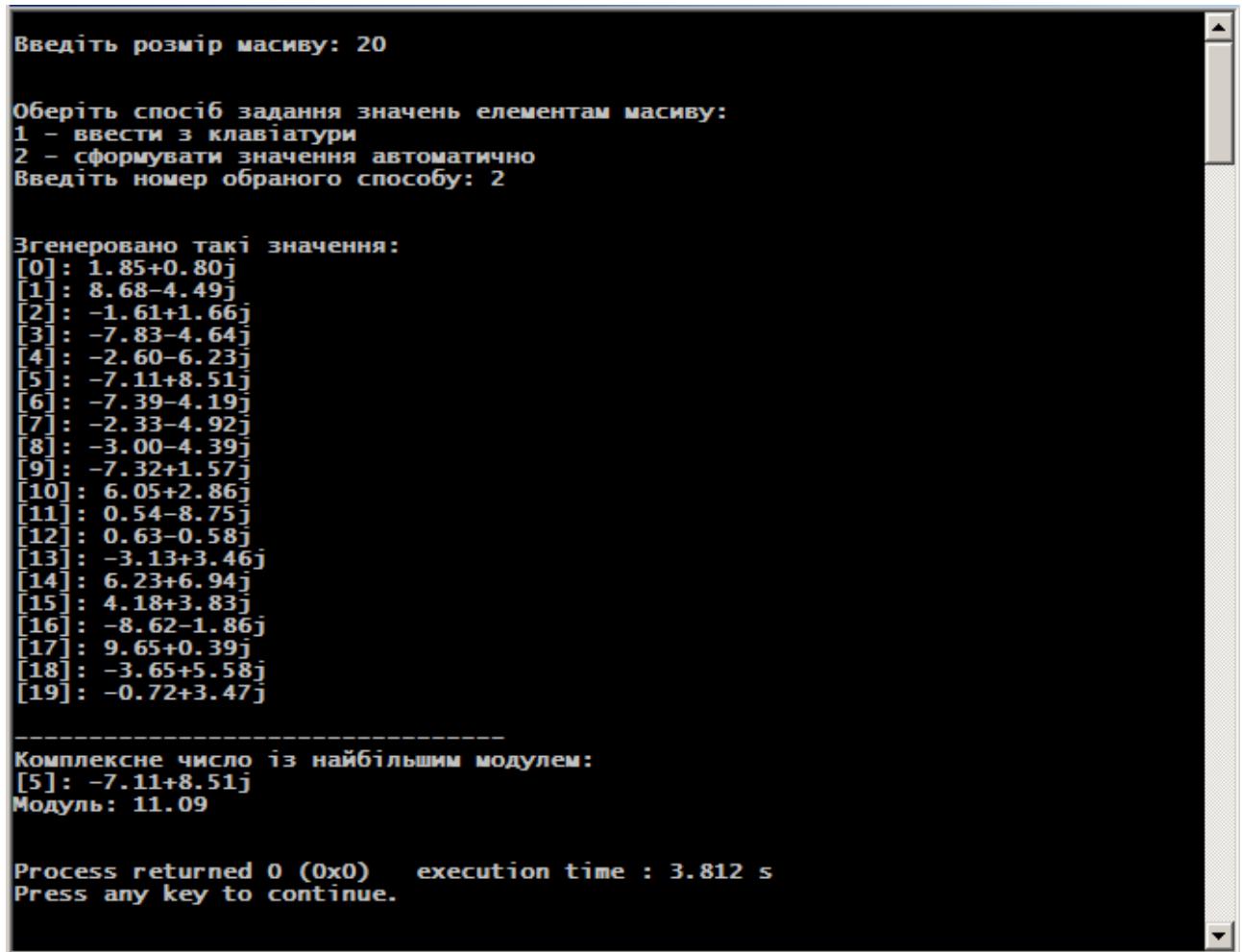
Process returned 0 (0x0)   execution time : 120.991 s
Press any key to continue.
```

Рис. 9.4 — Результат виконання програми пошуку комплексного числа із найбільшим модулем. Значення елементів масиву вводяться з клавіатури

На рис. 9.5 представлено результат виконання програми, коли обрався другий спосіб формування значень елементів масиву, а саме автоматичне генерування значень. В даному випадку розмір масиву було обрано рівним 20.

Для того, щоб кожного разу при новому запуску програми генерувався набір даних, який відмінний від попереднього запуску програми, необхідно підключити заголовочний файл `time.h`, і в код програми додати виклик функції `rand()`:

```
    srand(time(NULL));
```



```
Введіть розмір масиву: 20
Оберіть спосіб задання значень елементам масиву:
1 - ввести з клавіатури
2 - сформувати значення автоматично
Введіть номер обраного способу: 2

Згенеровано такі значення:
[0]: 1.85+0.80j
[1]: 8.68-4.49j
[2]: -1.61+1.66j
[3]: -7.83-4.64j
[4]: -2.60-6.23j
[5]: -7.11+8.51j
[6]: -7.39-4.19j
[7]: -2.33-4.92j
[8]: -3.00-4.39j
[9]: -7.32+1.57j
[10]: 6.05+2.86j
[11]: 0.54-8.75j
[12]: 0.63-0.58j
[13]: -3.13+3.46j
[14]: 6.23+6.94j
[15]: 4.18+3.83j
[16]: -8.62-1.86j
[17]: 9.65+0.39j
[18]: -3.65+5.58j
[19]: -0.72+3.47j

-----
Комплексне число із найбільшим модулем:
[5]: -7.11+8.51j
Модуль: 11.09

Process returned 0 (0x0)   execution time : 3.812 s
Press any key to continue.
```

Рис. 9.5 — Результат виконання програми пошуку комплексного числа із найбільшим модулем. Значення елементів масиву генеруються автоматично

При роботі із змінними та масивами структурного типу може застосовуватися операція `sizeof` аналогічним чином як вона застосовувалася до базових типів даних, а також до імен змінних і масивів базових типів даних.

Операція `sizeof` може знадобитися при роботі, наприклад, із динамічними масивами для визначення кількості байтів, що виділяється в пам'яті для деякого

структурного типу (як це показано в лістингу програми, що приведена вище), а також ця операція може застосовуватися при роботі із статичними масивами, коли в явному вигляді не задається розмір масиву при ініціалізації елементів масиву, а лише приводяться значення елементів масиву, як це показано в прикладі нижче, і в цьому випадку виникає необхідність визначити розмір масиву оперуючи даними про розмір в байтах, який виділений для всього масиву структур, а також розмір в байтах одного елементу масиву (або розміру в байтах структурного типу даних). В приведеному прикладі обчислюється розмір статичного масиву, використовуючи операцію `sizeof` з метою подальшого використання знайденої величини, яка застосовується у відповідному виразі-умови для контролю виходу за межі масиву.

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <math.h>

// Шаблон структури
struct complexNumber {
    double real;
    double imaginary;
};

int main()
{
    // Масив структур
    struct complexNumber X[ ] = {
        { 1.0, -0.5 }, // 1-й елемент
        { 2.0,  4.5 }, // 2-й елемент
        { 4.2, -3.8 }, // 3-й елемент
        {-0.7, -1.9 }, // 4-й елемент
        {-0.4,  0.2 }, // 5-й елемент
        { 5.1,  8.2 }, // 6-й елемент
        {-7.4,  0.9 }, // 7-й елемент
        { 2.3, -4.4 }, // 8-й елемент
        {-9.1,  0.3 }, // 9-й елемент
        { 0.6,  4.1 }, // 10-й елемент
    };
}
```

```

int i, sizeOfArray;

setlocale(LC_CTYPE, "Ukrainian");
printf("\nРозмір в Байтах структурного типу даних:\n");
printf("%u", sizeof( struct complexNumber ) );

printf("\n\nКількість Байт для статичного масиву структур:\n");
printf("%u", sizeof X );

printf("\n\nКількість Байт для одного елементу масиву:\n");
printf("%u", sizeof X[0] );

// Визначення розміру масиву:
sizeOfArray = sizeof X / sizeof (struct complexNumber);

printf("\n\nРозмір масиву: %d", sizeOfArray );
printf("\n\nЕлементи масиву:");
for( i = 0; i < sizeOfArray; i++ )
    printf("\n[%d]: %.2lf%+.2lfj", i, X[i].real, X[i].imaginary );

printf("\n\n");
return 0;
}

```

На рис. 9.6 показано результат виконання приведеної програми.

```

Розмір в Байтах структурного типу даних:
16
Кількість Байт для статичного масиву структур:
160
Кількість Байт для одного елементу масиву:
16
Розмір масиву: 10
Елементи масиву:
[0]: 1.00-0.50j
[1]: 2.00+4.50j
[2]: 4.20-3.80j
[3]: -0.70-1.90j
[4]: -0.40+0.20j
[5]: 5.10+8.20j
[6]: -7.40+0.90j
[7]: 2.30-4.40j
[8]: -9.10+0.30j
[9]: 0.60+4.10j

Process returned 0 (0x0)   execution time : 0.035 s
Press any key to continue.

```

Рис. 9.6 — Результати виконання програми, в якій визначається розмір статичного масиву структур за допомогою операції `sizeof`

При роботі із структурним типом даних можна створювати шаблон структури, що складається із полів (елементів) як базових типів даних так і структурних типів даних. В цьому випадку доводиться мати справу із вкладеними структурами [3—5, 7]. Такий підхід дозволяє створювати структурні типи даних на основі уже раніше створених структурних типів. Наприклад, розглянемо випадок, коли виникає необхідність певним чином описати піксель в цифровому кольоровому зображення. Нехай для зберігання інформації про колір використовується модель RGB, де колір пікселя формується шляхом змішування основних кольорів — червоного, зеленого та синього в певній пропорції. Таким чином, кожен піксель характеризується значеннями інтенсивності в червоному, зеленому та синьому каналах. Для представлення інтенсивності в червоному, зеленому та синьому каналах може бути використаний відповідний структурний тип, який буде складатися із трьох елементів (полів) — відповідне поле зберігає інформацію про інтенсивність у відповідному каналі. Нехай такий структурний тип буде називатися `color` і буде мати такий шаблон:

```
struct color {  
    int red;  
    int green;  
    int blue;  
};
```

Крім інформації про колір кожен піксель характеризується своїми координатами на цифровому зображення — це номер рядка та номер стовпця. Нехай нумерація рядків та стовпців починається із верхнього лівого кута. Для представлення даної інформації може бути використаний структурний тип даних, який складається із двох полів — перше поле призначено для зберігання номера рядка, друге поле — для зберігання номеру стовпця. Нехай для цих цілей використовується структурний тип даних, що має назву `coordinates`, і складається із двох полів — перше поле нехай називається `x`, друге поле називається `y`. Обидва поля мають тип даних `int`. В контексті задачі

роботи із пікселями на зображенні, поле `x` відповідає за номер стовпця, а поле `y` відповідає за номер рядка, в якому розміщується відповідний піксель на зображенні. Для якоїсь іншої задачі, створений структурний тип даних може використовуватися в іншому контексті, а поля з іменами `x` та `y` типу `int` можуть мати іншу інтерпретацію. Шаблон структури нехай буде мати такий вигляд:

```
struct coordinates {
    int x;
    int y;
}
```

Нехай структурний тип, що писує піксель, називається `pixel`. Шаблон структури `pixel` містить два поля — перше поле має ім'я `rgb` і призначено для зберігання інформації про колір, друге поле має ім'я `pos`, і призначено для зберігання позиції пікселя на цифрову зображення. Для прийнятого прикладу шаблон структури `pixel` буде мати такий вигляд:

```
struct pixel{
    struct color      rgb;
    struct coordinates pos;
};
```

Приклад програми, в якій використовуються вкладені структури представлено нижче:

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

// Шаблон структури
struct color {
    int red;
    int green;
    int blue;
};

// Шаблон структури
struct coordinates{
    int x;           // номер стовпця
    int y;           // номер рядка
};
```

```

// Шаблон структури
struct pixel{
    struct color      rgb;
    struct coordinates pos;
};

int main()
{
    // Ініціалізація структурної змінної
    struct pixel p = {
        {64, 128, 255}, // значення поля rgb
        {2, 5}           // значення поля pos
    };

    setlocale(LC_CTYPE, "Ukrainian");

    printf("\n\n Позиція пікселя:");
    printf("\n стовпець: %d", p.pos.x );
    printf("\n рядок: %d", p.pos.y );

    printf("\n\n Значення пікселя:");
    printf("\n red: %d", p.rgb.red);
    printf("\n green: %d", p.rgb.green );
    printf("\n blue: %d", p.rgb.blue );

    printf("\n\n");
    return 0;
}

```

В наведеному прикладі змінна **p** — це змінна структурного типу **struct pixel**. Відповідно до шаблону структури **struct pixel**, структурна змінна **p** складається із двох полів — **rgb** та **pos**. Кожне із цих полів є структурним типом, шаблони яких описано в програмі. Таким чином, при ініціалізації змінної **p** потрібно у відповідній послідовності, яка визначена в шаблоні структури **struct pixel** та з врахуванням шаблонів вкладених структурах, задати значення для поля **rgb** та поля **pos** — це значення **{64, 128, 255}** та **{2, 5}** відповідно.

При виведенні на екран значення змінної **p** структурного типу **struct pixel**, наприклад, при відображені значень інтенсивності відповідного

кольору, потрібно спочатку звернутися до поля `rgb` структурного типу `struct color`, а потім вже до поля `red`, `green` та `blue` структури `struct color` відповідно:

```
p.rgb.red  
p.rgb.green  
p.rgb.blue
```

При відображені на екрані значення позиції пікселя спочатку необхідно звернутися до поля `pos` структурного типу `struct coordinates`, а потім вже до поля `x` та `y` структури `struct coordinates` відповідно:

```
p.pos.x  
p.pos.y
```

До структурних змінних не можна безпосередньо застосовувати операції відношення, наприклад, операцію «більше», «менше» та ін., в разі необхідності порівняння двох структурних змінних, наприклад, на предмет їх рівності, потрібно виконувати окремо перевірку кожного поля відповідних структурних змінних, причому якщо ці поля є структурами, то потрібно переходити до порівняння полів вже у вкладених структурах і т.д. [3, 4].

Наприклад, якщо відповідно до вище вказаних шаблонів структур задано дві структурні змінні з іменами `p` та `d`, які ініціалізуються в програмі таким чином:

```
struct pixel p = { {64, 128, 255}, {2, 5} }, d = { {0, 0, 0}, {0, 0} };
```

То наступні варіанти застосування операцій відношення є **помилковими**:

```
p > d  
p.pos > d.pos  
p.rgb > d.rgb
```

Щоб порівняти між собою змінні `p` та `d` коректним способом — потрібно перейти до рівня полів, які сформовані із базових типів:

```

p.rgb.red > d.rgb.red
p.rgb.green > d.rgb.green
p.rgb.blue > d.rgb.blue

p.pos.x > d.pos.x
p.pos.y > d.pos.y

```

Значення структурної змінної можна безпосередньо присвоювати структурній змінній цього ж типу. В наступному прикладі змінній `d` присвоюється значення змінної `p`:

```
d = p;
```

При цьому полям змінної `d` присвоюються значення відповідних полів змінної `p`.

9.3. Принципи написання функцій по обробці структур

При реалізації відповідних алгоритмів виникає необхідність окремі пункти завдання реалізовувати у вигляді функцій, причому в якості параметрів ці функції можуть отримувати структури. Крім того, структура може бути типом даних результату, що повертається функцією. Загалом, написання функцій, які отримують аргументи структурного типу даних та повертають значення структурного типу даних, багато в чому відповідають тим підходам та принципам, які були розглянуті раніше при передачі у функцію змінних базових типів даних, а також вказівників, масивів та символічних рядків. Враховуючи, що структурний тип даних дозволяє описати в програмі деякий об'єкт над яким потрібно виконувати певну обробку, то саму обробку можна реалізувати у вигляді окремих функцій. В якості прикладу розглянемо варіант написання функцій, які передбачають оперування даними структурного типу, який був раніше розглянутий для зберігання комплексних чисел.

Створимо структурний тип даних, який буде крім дійсної та уявної частини комплексного числа, також містити поле, яке призначене для

зберігання модуля комплексного числа. Раніше при роботі із комплексними числами був створений такий шаблон структури:

```
struct complexNumber {  
    double real;  
    double imaginary;  
};
```

Створимо структурний тип, який буде містити в якості одного із полів вище вказану структуру (тобто одне із полів буде представляти собою вкладену структуру), а друге поле — буде представляти собою поле типу `double` для зберігання модуля комплексного числа. Нехай створений структурний тип буде називатися `struct cValue`:

```
struct cValue {  
    struct complexNumber num;  
    double mod;  
};
```

Таким чином, поле `num` — це структура `struct complexNumber`, тобто поле `num` в свою чергу складається із полів `real` та `imaginary` типу `double`. Поле `num` призначено для зберігання дійсної та уявної частини комплексного числа.

Поле `mod` має тип даних `double` і призначено для зберігання модуля комплексного числа, дійсна та уявна частини якого зберігаються в полі `num`.

В програмі реалізуються наступні функції:

1. Функція `getComplexNumber()` — використовується для зчитування дійсної та уявної частини комплексного числа з клавіатури. Функція має такий прототип:

```
struct cValue getComplexNumber( void );
```

Функція не отримує ніяких параметрів і повертає значення структурного типу `struct cValue`.

2. Функція `printComplexNumber()` призначена для виведення на екран в консольне вікно комплексного числа у форматі $r\pm sj$, де r — це дійсна частина

числа, s — уявна частина числа, j — уявна одиниця. Функція має такий прототип:

```
void printComplexNumber( const struct cValue );
```

Функція `printComplexNumber()` має тип даних `void`, таким чином вона не повертає ніякого значення. Функція в якості параметра отримує структурну змінну типу `struct cValue`, а службове слово `const` забезпечує неможливість зміни значення полів структурної змінної в ході роботи даної функції, тобто функція сприймає аргумент, який вона отримує, як константний.

3. Функція `addTwoComplexNumbers()` призначена для знаходження суми двох комплексних чисел. Прототип функції:

```
struct cValue addTwoComplexNumbers( const struct cValue, const struct cValue );
```

Функція `addTwoComplexNumbers()` отримує два параметри. В якості параметрів виступають дві структурні змінні. Службове слово `const` як і в попередній функції показує, що параметри, які передаються у функцію повинні розглядатися як константи. Функція розраховує значення суми двох комплексних чисел і повертає результат в точку виклику. Тип даних результату — це структурний тип `struct cValue`.

4. Функція `calculateModule()` використовується для розрахунку модуля комплексного числа. Прототип функції такий:

```
void calculateModule( struct cValue * );
```

Функція має тип даних `void`, тобто не повертає результат. Функція в якості параметру отримує вказівник на структуру — це необхідно щоб функція змогла, розрахувавши значення модуля комплексного числа, присвоїти обраховане значення полю `mod` тієї структурної змінної, вказівник на яку буде переданий у функцію в якості параметру.

5. Функція `printModul()` використовується для виведення на екран значення модуля комплексного числа. Данна функція має такий прототип:

```
void printModul( const struct cValue );
```

Функція отримує структурну змінну і виводить на екран в консольне вікно значення поля `mod`. Оскільки функція повинна лише вивести значення переданого їй параметра на екран і не змінювати це значення, з цією метою використано службове слово `const`.

Вказані функції представлені для ілюстрації використання функцій при роботі із структурами. Лістинг програми представлено нижче:

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <math.h>

// Шаблон структури
struct complexNumber {
    double real;
    double imaginary;
};

// Шаблон структури
struct cValue {
    struct complexNumber num;
    double mod;
};

//-----
// прототипи функцій
//-----

struct cValue getComplexNumber( void );
void printComplexNumber( const struct cValue );
struct cValue addTwoComplexNumbers( const struct cValue, const struct cValue );
void calculateModule( struct cValue * );
void printModul( const struct cValue );

//-----
```

```

int main()
{
    // a, b та c – структурні змінні
    struct cValue a, b, c;

    setlocale(LC_CTYPE, "Ukrainian");

    // Введення значень змінних a та b
    printf("\nВведіть перше комплексне число.");
    a = getComplexNumber();

    printf("\nВведіть друге комплексне число.");
    b = getComplexNumber();

    // Виведення на екран заданих значень
    // для контролю правильності їх введення
    printf("\n\n-----");
    printf("\nВи задали наступні значення:");
    printf("\na = ");
    printComplexNumber( a );
    printf("\nb = ");
    printComplexNumber( b );

    // Знаходження суми комплексних чисел a та b
    // Результат обрахунку зберігається в змінній c
    c = addTwoComplexNumbers( a, b );

    // Виведення на екран значення змінної c
    printf("\n\nОбрахунок    c = a + b");
    printf("\nc = ");
    printComplexNumber( c );

    // Обрахунок модуля комплексного числа
    calculateModule( &a );
    calculateModule( &b );
    calculateModule( &c );
}

```

```

// Виведення на екран результатів обрахунку модуля

printf("\n\nРозраховане значення модуля комплексного числа:");
printf("\nМодуль (a) = ");
printModul( a );
printf("\nМодуль (b) = ");
printModul( b );
printf("\nМодуль (c) = ");
printModul( c );

printf("\n\n");
return 0;
}

//-----
//----- Опис функцій -----
//


// Функція для введення з клавіатури
// дійсної та уявної частини комплексного числа

struct cValue getComplexNumber( void )
{
    struct cValue temp;

    printf("\nДійсна частина: ");
    scanf("%lf", &temp.num.real );
    printf("Уявна частина: ");
    scanf("%lf", &temp.num.imaginary );

    return temp;
}

//-----

// Функція виводить на екран в консольне вікно комплексне число
// у форматі r+sj, де r - дійсна частина, s - уявна частина,
// j - уявна одиниця

void printComplexNumber( const struct cValue x )
{
    printf("%.2lf%+.2lfj", x.num.real, x.num.imaginary );
}

//-----

```

```

// Функція розраховує суму двох комплексних чисел

struct cValue addTwoComplexNumbers(const struct cValue x, const struct cValue y)
{
    struct cValue temp;

    temp.num.real      = x.num.real + y.num.real;
    temp.num.imaginary = x.num.imaginary + y.num.imaginary;

    return temp;
}

//-----

// Функція розраховує модуль комплексного числа, використовуючи
// при цьому вказівник на структуру, щоб мати змогу змінити
// значення поля mod

void calculateModule( struct cValue * ptr )
{
    ptr->mod = sqrt( pow(ptr->num.real, 2) + pow(ptr->num.imaginary, 2) );
}

//-----

// Функція виводить на екран в консольне вікно значення модуля
// комплексного числа

void printModul( const struct cValue x)
{
    printf("%.2lf", x.mod );
}

//-----
```

На рис. 9.7 приведено приклад виконання програми із результатами, що відображаються в консольному вікні.

На основі представлених варіантів реалізацій функцій по обробці структур, можна написати ряд функцій, які б дозволяли розширити набір математичних дій над комплексними числами, а також забезпечити можливість

опрацювання масивів структур, які призначені для зберігання комплексних чисел.

```
Введіть перше комплексне число.  
Дійсна частина: -4.5  
Уявна частина: 2.0  
  
Введіть друге комплексне число.  
Дійсна частина: 9.75  
Уявна частина: -5.5  
  
-----  
Ви задали наступні значення:  
a = -4.50+2.00j  
b = 9.75-5.50j  
  
Обрахунок c = a + b  
c = 5.25-3.50j  
  
Розраховане значення модуля комплексного числа:  
Модуль(a) = 4.92  
Модуль(b) = 11.19  
Модуль(c) = 6.31  
  
Process returned 0 (0x0)   execution time : 24.896 s  
Press any key to continue.
```

Рис. 9.7 — Результати виконання програми

За допомогою структур існує можливість створювати типи даних, які б більш адекватно та органічно узгоджувалися із тими даними над якими потрібно виконувати обробку в ході реалізації відповідних алгоритмів [3—7].

9.4. Питання для самоконтролю

1. Поясніть що таке шаблон структури? Навіщо використовуються структури?
2. Що таке елемент структури? В чому відмінність між елементом структури і полем структури?
3. Яким чином можна визначити об'єм пам'яті в байтах, який займає структура?
4. Приведіть приклад де було б доцільно використовувати структури?
5. Поясніть яким чином можна дістатися до значення поля структури використовуючи вказівник на структуру?

СПИСОК ЛІТЕРАТУРИ

1. Tanenbaum A. Structured Computer Organization. Sixth Edition / A. Tanenbaum. — Pearson, 2013. — 776 p.
2. Harris D. Digital Design and Computer Architecture. Second Edition / D. Harris, S.Harris. — Morgan Kaufmann, 2012. — 720 p.
3. Prata S. C Primer Plus. Sixth Edition / S. Prata. — Addison-Wesley, 2013. — 1067 p.
4. Deitel P. C: How to Program. With an introduction to C++. Eighth Edition / P. Deitel, H. Deitel. — Pearson, 2015. — 1005 p.
5. Schildt H. C: The Complete Reference / H. Schildt. — McGraw Hill, 2000. — 805 p.
6. Habrison S. C: A Reference Manual / S. Habrison — Pearson, 2002. — 533.
7. Шпак З.Я. Програмування мовою С: Навчальний посібник / З.Я. Шпак. — Оріяна-Нова, 2006. — 432.
8. Татарчук Д.Д. Програмування мовами С та С++: навч. посіб. / Д.Д. Татарчук, Ю.В. Діденко. — Київ, 2012. — 112 с.
9. Інформатика. Основи програмування та алгоритми. Мова програмування С. Лабораторний практикум [Електронний ресурс]: навч. посіб. для студ. спеціальності 172 / уклад.: С. В. Вишневий, П. Ю. Катін, С. В. Крилов. — Електронні текстові дані (1 файл: 3,3 Мбайт). — Київ: КПІ ім. Ігоря Сікорського, 2022. — 221 с.