МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

А.Г. Кривохата, О.В. Кудін

ОСНОВИ ПРОГРАМУВАННЯ

Навчальний посібник
для здобувачів ступеня вищої освіти бакалавра
спеціальності «Інженерія програмного забезпечення»
освітньо-професійної програми «Програмна інженерія»

Затверджено
вченою радою ЗНУ
Протокол № 7 від 29.12.2025

Запоріжжя
2025

УДК 004.4(075.8)
К821

Кривохата А.Г., Кудін О.В. Основи програмування : навчальний посібник для здобувачів ступеня вищої освіти бакалавра спеціальності «Інженерія програмного забезпечення» освітньо-професійної програми «Програмна інженерія». Запоріжжя : Запорізький національний університет, 2025. 107 с.

У навчальному посібнику подано теоретичний матеріал до тем курсу «Основи програмування». Значна увага приділяється організації обчислень лінійної, розгалуженої і циклічної структур. Розглянуто основні правила процедурного програмування, обробки масивів, динамічного розподілу пам’яті, роботи з файлами.

До кожної теми надано практичні завдання, які дають змогу застосувати отримані знання до розв’язання конкретних задач, тестові завдання для самоперевірки, контрольні запитання, відповіді на які сприятимуть засвоєнню теоретичного матеріалу.

Для здобувачів ступеня вищої освіти бакалавра спеціальності «Інженерія програмного забезпечення» освітньо-професійної програми «Програмна інженерія».

Рецензент
О.С. Пшенична, кандидат педагогічних наук, доцент, в.о. зав. кафедри комп’ютерних наук

Відповідальний за випуск
А.О. Лісняк, кандидат фізико-математичних наук, доцент, завідувач кафедри програмної інженерії

Зміст

Вступ 5
Тема 1. Програмне забезпечення та мови програмування 7
1.1 Етапи розв'язання задач на комп'ютері 7
1.2 Основні парадигми програмування 8
1.3 Загальні відомості про системи числення 8
1.4 Перетворення числа з двійкової системи у вісімкову та шістнадцяткову 10
1.5 Перетворення числа з десяткової системи в іншу позиційну систему числення 11
1.6 Перетворення числа з двійкової (вісімкової, шістнадцяткової) у десяткову систему числення 12
1.7 Основні поняття алгоритмізації 12
Тема 2. Елементи мови програмування С 17
2.1 Елементи мови С 17
2.2 Структура С-програми 18
2.3 Типи даних 20
2.4 Змінні 21
2.5 Базові операції над даними 23
2.6 Перетворення типів 24
2.7 Форматоване виведення даних 24
2.8 Форматоване введення даних 27
2.7 Програми з лінійною структурою 27
2.8 Основні заголовні файли С 29
Тема 3. Програмування розгалужених процесів 32
3.1 Логічні операції в мові програмування С 32
3.3 Оператор розгалуження `if` 34
3.4 Тернарний оператор 36
3.5 Оператор вибору варіантів `switch` 37
Тема 4. Програмування циклічних процесів 42
4.1 Цикли з параметром `for()` 42
4.2 Цикл з передумовою 47
4.3 Цикл з післяумовою 48
4.4 Оператори `break i continue` 49
Тема 5 Масиви 53
5.1 Основні поняття про масиви 53
5.2 Одновимірні масиви 54
5.3 Генерування послідовності псевдовипадкових чисел 56
5.4 Двовимірні масиви 58
Тема 6. Вказівники. Створення динамічного масиву 63
6.1 Основні поняття. Вказівники у С/С++ 63

6.2 Операції із вказівниками ... 64
6.3 Арифметика вказівників ... 66
6.4 Вказівники на вказівники ... 68
6.5 Вказівники на функції ... 68
6.6 Динамічні масиви в C/C++ ... 69
Тема 7. Символи та рядки ... 74
7.1 Представлення символів в С ... 74
7.2 Представлення рядків в С ... 76
7.3 Функції обробки рядків ... 77
Тема 8. Функції користувача ... 83
8.1 Поняття про функцію та її складові ... 83
8.2 Функції типу void ... 85
8.3 Функції, що повертають значення ... 86
8.4 Передавання параметрів у функцію ... 87
8.5 Рекурсивні функції ... 91
Тема 9. Файли ... 97
9.1 Основні функції для роботи з файлами в С. ... 97
9.2 Загальна схема роботи з файлом ... 97
9.3. Запис та читання інформації з файлу ... 99
Додаток А Перелік зарезервованих слів мови програмування С ... 103
Рекомендована література ... 105
Використана література ... 106
Інформаційні джерела ... 107

Вступ

В умовах стрімкої цифрової трансформації та інтеграції інформаційних технологій у всі сфери людської діяльності, дисципліна, присвячена вивченню основ програмування, набуває стратегічної важливості. Актуальність цього курсу обумовлена об'єктивною потребою ринку праці у висококваліфікованих ІТ-фахівцях, які здатні не лише використовувати наявні цифрові інструменти, але й ефективно створювати, адаптувати та підтримувати складні програмні продукти.

Дане навчальне видання являє собою комплексний та структурований посібник, покликаний забезпечити читача ґрунтовними теоретичними знаннями та розвинути практичні навички, необхіднми для успішної кар'єри у сфері розробки програмного забезпечення. У посібнику детально викладено теоретичний матеріал, який одразу закріплюється на прикладах створення повноцінних програмних проєктів мовою С.

Метою вивчення навчальної дисципліни «Основи програмування» є: оволодіння теоретичними знаннями та набуття навичок з програмування та алгоритмізації, які дають можливість вирішувати задачі реалізації різних видів програмного забезпечення. Опанування даного курсу передує вивченню курсів «Об'єктно-орієнтовне програмування», «Алгоритми і структури даних», «Організація комп'ютерних мереж», «Функціональне програмування».

Основними **завданнями** вивчення дисципліни «Основи програмування» є: засвоєння основних понять програмування, базових структур даних; ознайомлення з основними парадигмами програмування; опанування основних алгоритмів розв'язку типових задач та їх реалізація за допомогою конкретних мов програмування; набуття навичок розробки власних програм на мовах програмування для вирішення прикладних задач.

Згідно з вимогами освітньої-професійної програми здобувачі освіти мають досягти наступних компетентностей та програмних результатів навчання:

- здатність до абстрактного мислення, аналізу та синтезу;
- здатність накопичувати, обробляти та систематизувати професійні знання щодо створення і супроводження програмного забезпечення та визнання важливості навчання протягом всього життя;
- здатність обґрунтовано обирати та освоювати інструментарій з розробки та супроводження програмного забезпечення;
- здатність до алгоритмічного та логічного мислення;
- аналізувати, цілеспрямовано шукати і вибирати необхідні для вирішення професійних завдань інформаційно-довідникові ресурси і знання з урахуванням сучасних досягнень науки і техніки;
- уміння вибирати та використовувати відповідну задачі методологію створення програмного забезпечення;

- знати і застосовувати на практиці фундаментальні концепції, парадигми і основні принципи функціонування мовних, інструментальних і обчислювальних засобів інженерії програмного забезпечення;
- вміти застосовувати методи компонентної розробки програмного забезпечення.

Структура та вміст навчального посібника відповідають змісту силабуса дисципліни «Основи програмування».

Навчальне видання охоплює всі ключові аспекти, необхідні для формування інженерної бази. Розглянуто основи алгоритмізації та принципи побудови програмної логіки. Посібник детально розкриває створення лінійних, розгалужених та циклічних алгоритмів. Окрема увага приділяється ефективній роботі з даними. Це включає вивчення статичних та динамічних масивів, що вимагає розуміння вказівників та механізмів керування оперативною пам'яттю. Також докладно розглянуто методи обробки рядкових даних як особливого типу масивів символів. Подано основи процедурного програмування (робота з функціями, передача параметрів) для створення модульного та структурованого коду. Вивчення взаємодії з файлами завершує курс, навчаючи програму зберігати та отримувати дані з зовнішньої файлової системи. До кожної теми наведено тестування для самоконтролю, контрольні запитання, переліки рекомендованої та використаної літератури, посилання на джерела, які можна використовувати при розробці програм

Тема 1. Програмне забезпечення та мови програмування

План

1. Етапи розв’язання задач на комп’ютері.
2. Основні парадигми програмування.
3. Загальні відомості про системи числення.
4. Перетворення числа з двійкової системи у вісімкову та шістнадцятькову.
5. Перетворення цілого числа з десяткової системи в будь-яку іншу позиційну систему числення.
6. Перетворення числа з двійкової (вісімкової, шістнадцятькової) у десяткову систему числення.
7. Основні поняття алгоритмізації.

1.1 Етапи розв’язання задач на комп’ютері

Робота з розв’язання прикладної задачі з використанням комп’ютера проходить у кілька етапів, кожен із яких важливий по-своєму, і тільки правильне виконання кожного з етапів призводить до розв’язання задачі у цілому.

Перед виконанням будь-якого проекту доцільно змоделювати хід виконання і спрогнозувати результати роботи. У цьому може допомогти побудова інформаційної моделі об’єкта-оригіналу або явища (процесу), що досліджується. У більшості випадків це дозволяє швидко і з невеликими витратами розрахувати результати виконання масштабної реальної задачі.

Основними етапами є:

1. **Постановка задачі** передбачає визначення умов задачі (що дано, що необхідно, які є припущення, які результати та в якому вигляді повинні бути отримані), проведення змістовного аналізу з уточненням мети розв’язання задачі. У межах цього етапу визначаються умови отримання результатів і те, які результати вважатимуться вірними.
2. **Побудова математичної моделі**, на цьому етапі змістовний опис задачі замінюється математичною моделлю та обґрунтовано обирається метод розв’язання.
3. **Складання алгоритмів** на основі методу, що обрано.
4. **Опис алгоритму** мовою програмування.
5. **Тестування** та налагодження програм.
6. **Остаточне виконання програми**, аналіз результатів та експлуатація програми.

Перед виконанням будь-якого проекту доцільним є моделювання ходу виконання і прогноз результатів роботи. У цьому може допомогти побудова інформаційної моделі об’єкта або процесу, що досліджується.

1.2 Основні парадигми програмування

Програма являє собою опис тієї послідовності дій, яку повинен виконати комп'ютер. Цей опис з боку машини виглядає як послідовність ***машинних команд*** – команд, які машина вміє виконувати, оскільки механізми їх виконання реалізовано в електронній схемі комп'ютера. Сукупність машинних команд складає так звану ***машинну мову***, або ***мову низького рівня***. Втім, значно вигідніше підлаштувати комп'ютер під людські потреби з метою збільшення продуктивності праці програміста. Для цього призначені мови програмування високого рівня. З одного боку, мова програмування високого рівня розробляється так, щоб текст програми ясно відображав її сенс. З іншого боку, мова програмування високого рівня проектується таким чином, щоб будь-який текст можна було б перекласти з неї на машинну мову. Такий переклад робить не людина, а програма, яка називається ***транслятор***.

Програма у вигляді текстового файла проходить 3 етапи опрацювання:

- Препроцесорне перетворення тексту;
- Компіляція (перевіряє текст на наявність помилок та перетворює текст програми у набір машинних команд);
- Компонування (редагування зв'язків або збірка).

Різні задачі програмування можна вирішувати по-різному: може знадобитися написати функцію, створити окремий клас із методами тощо. Усі такі варіанти об'єднуються в різні підходи програмування, які також називаються ***парадигмами***. Основними парадигмами програмування є декларативне та імперативне.

Декларативне програмування описує, який результат необхідно отримати, замість опису послідовності дій для отримання цього результату. До декларативної парадигми належать функційне та логічне програмування. Прикладами є такі мови програмування **`SQL`**, **`HTML`**, **`CSS`**, **`Prolog`**, **`Haskell`**.

Імперативне програмування описує процес отримання результатів як послідовність інструкцій зміну стану програми. До імперативної парадигми належать структурне, процедурне, модульне (об'єктне), об'єктно-орієнтоване програмування. Прикладами є такі мови програмування **`C`**, **`C++`**, **`Java`**, **`Python`** тощо.

1.3 Загальні відомості про системи числення

Система числення – це сукупність прийомів та правил, за якими числа записуються та читаються.

Існують **позиційні** та **непозиційні** системи числення.

У *непозиційних* системах числення вага цифри (тобто той вклад, який вона вносить у значення числа) не залежить від її позиції в запису числа. Наприклад, у римській системі числення в числі XXXII (тридцять два) вага цифри X у будь-якій позиції дорівнює десяти.

В *позиційних* системах числення вага кожної цифри змінюється в залежності від її положення (позиції) в послідовності цифр, що зображують число.

Приклад 1.1 В числі 757,7 перша сімка позначає 7 сотен, друга – 7 одиниць, а третя – 7 десятих частин одиниці.

Сам запис числа 757,7 означає скорочений запис виразу:

$$700 + 50 + 7 + 0{,}7 = 7*10^2 + 5*10^1 + 7*10^0 + 7*10^{-1} = 757{,}7.$$

Будь-яка позиційна система числення характеризується своєю основою.

Основа позиційної системи числення – кількість різних цифр, що використовуються для зображення чисел у даній системі числення.

За основу системи числення можна взяти будь-яке натуральне число – два, три, чотири та ін. Отже існує *нескінченна множина позиційних систем*: двійкова, трійкова та ін. Запис чисел в кожній із систем числення з основою q означає скорочений запис виразу:

$$a_{n-1}\, q^{n-1} + a_{n-2}\, q^{n-2} + \ldots + a_1\, q^1 + a_0\, q^0 + a_{-1}\, q^{-1} + \ldots + a_{-m}\, q^{-m},$$

де a_i – цифри системи числення; n і m – кількість цілих і дробових розрядів відповідно.

Далі наведемо запис в системах числення перших шістнадцяти чисел цілих чисел.

Таблиця 1.1 – Цілі числа в різних системах числення

Системи числення			
Десяткова	Двійкова	Вісімкова	Шістнадцяткова
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Двійкова система числення має низку переваг відносно інших систем числення:

- Для її реалізації потрібні технічні пристрої з двома стійкими станами (є напруга – відсутня напруга), а не, наприклад, з десятьма, – як в десятковій.

- Можливість застосування апарату булевої алгебри для виконання логічних перетворень інформації.

Недолік двійкової системи – швидке зростання кількості розрядів, необхідних для запису чисел.

Головна роль шістнадцяткової системи числення (Hex) – зручне і компактне представлення довгого, складно читаного двійкового коду, який використовує комп'ютер. Оскільки $16 = 2^4$, це означає, що **одна шістнадцяткова цифра** точно відповідає **чотирьом двійковим бітам** (це називається «нібл» - nibble). Також варто зазначити, що переведення з двійкової в шістнадцяткову систему та навпаки є прямим і не вимагає складних обчислень. Наприклад, двійкове число $101_2 = D_{16}$. Ця простота робить Hex ідеальною мовою для розробників. Ключова функція шістнадцяткової системи в програмуванні – це представлення **адрес пам'яті** та **значень байтів**:

1. **Адресація пам'яті:** Оперативна пам'ять (RAM) організована як послідовність байтів, і кожен байт має свою унікальну адресу. У сучасних 64-розрядних системах адреса – це послідовність із 64 бітів.
2. **Компактність:** Замість того, щоб писати довжелезну двійкову адресу (наприклад, 64 нулі та одиниці), яка легко призводить до помилок, програмісти використовують Hex. 64 біти ідеально стискаються до **16 шістнадцяткових цифр**. Наприклад, адресу `000100101110` набагато зручніше записати як `0x12EF`. Префікс `0x` в мові C вказує компілятору, що далі йде шістнадцяткове число.
3. **Відлагодження (Debugging):** При відлагодженні програми або аналізі дампів пам'яті (memory dumps), Hex дозволяє швидко знаходити та ідентифікувати конкретні місця в пам'яті, де зберігаються дані або виконуваний код.

1.4 Перетворення числа з двійкової системи у вісімкову та шістнадцяткову

Числа у вісімковій та шістнадцятьковій системах читаються так само легко, як десяткові, але вимагають відповідно в три (вісімкова) та в чотири (шістнадцяткова) рази менше розрядів, ніж у двійковій системі.

Для **перетворення вісімкових та шістнадцяткових чисел у двійкову** систему необхідно кожну цифру вихідного числа замінити еквівалентною їй двійковою **тріадою** (трійкою цифр) або **тетрадою** (четвіркою цифр) відповідно до таблиці 1.1.

Для того, щоб перевести число з двійкової системи у вісімкову або шістнадцяткову системи числення, його потрібно розбити вліво і вправо від

коми на *тріади* (для вісімкової) або *тетради* (для шістнадцяткової) і кожну таку групу замінити відповідною їй вісімковою (шістнадцятковою) цифрою.

Приклад 1.2 Перевести число 10101001,10111 у вісімкову та шістнадцяткову системи числення.

Розбиваємо число вліво і вправо від коми (на рисунку напрям розбиття показано жовтими стрілками) на тріади (для перетворення у вісімкову систему числення):

$$10101001{,}10111_2 = \underline{10}\ \underline{101}\ \underline{001}\ ,\ \underline{101}\ \underline{110} = 251{,}56_8$$

2 5 1 5 6

Відповідно до таблиці 1.1 кожну тріаду замінили на відповідну їй вісімкову цифру.

Для перетворення числа у шістнадцяткову систему числення розбиваємо його вліво і вправо від коми (на рисунку напрям розбиття показано жовтими стрілками) на тетради:

$$10101001{,}10111_2 = \underline{1010}\ \underline{1001}\ ,\ \underline{1011}\ \underline{1000} = A9{,}B8_{16}$$

A 9 B 8

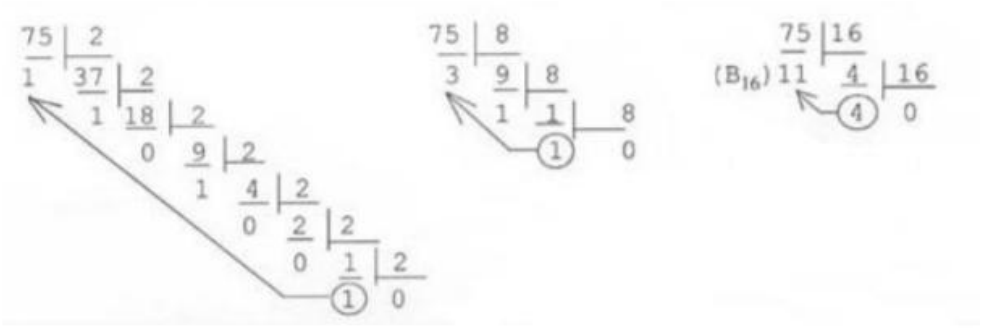
1.5 Перетворення числа з десяткової системи в іншу позиційну систему числення

Для перетворення цілого десяткового числа ***N*** в систему числення з основою ***q*** необхідно ***N*** поділити з остачею («націло») на ***q***, яке записано в тій же десятковій системі. Далі неповну частку, яку отримано від такого ділення, потрібно знову розділити з остачею на ***q*** і т.д. поки остання отримана неповна частка не дорівнюватиме нулю. Представленням числа ***N*** в новій системі числення буде послідовність остач від ділення, які зображено однією ***q***-ічною цифрою та записаних у порядку оберненому до порядку їх отримання.

Приклад 1.3 Перевести число 75 із десяткової системи числення у двійкову, вісімкову, шістнадцяткову.

Для того, щоб перевести число із десяткової системи числення у двійкову необхідно поділити число з остачею на 2. Потім неповну частку, отриману від такого ділення знову розділити на 2. Ділити необхідно доти поки остання отримана неповна частка не дорівнюватиме нулю. Представленням числа у двійковій системі числення буде послідовність остач від ділення, записаних у порядку оберненому до їх отримання. Аналогічно здійснюється перетворення із десяткової у будь-яку іншу систему числення.

Для того, щоб не перевантажувати запис у діленні стовпчиком записуватимемо отримані остачі.



Таким чином, $75_{10}= 1001011_2=113_8=4B_{16}$

1.6 Перетворення числа з двійкової (вісімкової, шістнадцяткової) у десяткову систему числення

Перетворення в десяткову систему числа $\boldsymbol{x}$, яке записано у $\boldsymbol{q}$-ічній системі числення ($\boldsymbol{q}$ = 2, 8 або 16) у вигляді $x_q = (a_n a_{n-1} \dots a_0, a_{-1} a_{-2} \dots a_{-m})_q$ зводиться до обчислення значення многочлена

$$x_{10} = a_n q^n + a_{n-1} q^{n-1} + \cdots + a_0 q^0 + a_{-1} q^{-1} + \cdots + a_{-2} q^{-2} + \cdots + a_{-m} q^{-m}$$

Приклад 1.4 Перевести числа $1011,1_2$; $276,5_8$; $1F3_{16}$ у десяткову систему числення

3 2 1 0 –1
1 0 1 1, $1_2=1*2^3+0*2^2+1*2^1+1*2^0+1*2^{-1}=11,5_{10}$

2 1 0 –1
2 7 6, $5_8=2*8^2+7*8^1+6*8^0+5*8^{-1}=190,625_{10}$

2 1 0
1 F $3_{16}=1*16^2+15*16^1+3*16^0=499_{10}$

1.7 Основні поняття алгоритмізації

Алгоритм – це зрозумілий і точний припис (вказівка) виконавцю здійснити визначену послідовність дій над вихідною інформацією для розв’язання поставленої задачі.

Таким чином, алгоритм визначає певні правила перетворювання інформації, тобто визначає певну послідовність операцій опрацювання даних для отримання розв’язку задачі.

Властивості алгоритмів:

- **Дискретність**. Ця властивість означає, що алгоритм може бути розділено на окремі досить прості дії, причому виконання кожного кроку починається після завершення попереднього.

- **Визначеність**. Однозначність виконання кожного кроку. Алгоритм не має припускати довільного трактування виконавцем.

- **Результативність.** (виконуваність). Алгоритм повинен надавати можливість отримання рішення за скінченну кількість кроків.

- **Масовість**. Це придатність для розв'язання багатьох або навіть всіх задач даного типу при різних вихідних даних.

- **Інваріантність**. Алгоритм має бути складено таким чином, щоб він, в ідеальному випадку міг бути виконаним різними користувачами: різними ЕОМ, у різних середовищах, різними людьми.

Існують різні способи запису алгоритмів. Далі наведемо короткий опис кожного з них.

Словесний запис орієнтовано на виконавця – людину. При такому способі команди записуються природною мовою та нумеруються.

Наприклад, розглянемо алгоритм Евкліда для пошуку найбільшого спільного дільника.

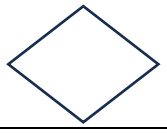
1. Задати два числа.
2. Якщо числа рівні, то відповідь дорівнює одному з чисел, інакше перейти до п.3
3. Визначити число, яке є більшим за друге.
4. Замінити більше число на частку більшого та меншого числа.
5. Перейти до п.2

Основна перевага такого способу представлення – зрозумілість. До недоліків можна віднести: неоднозначність, надлишковість, відсутність наочності зв'язків.

Графічне зображення алгоритма, згідно затвердженим стандартам називається **блок-схемою**. При такому способі кожна дія записується всередині блоків. Блоки з'єднуються лініями, які вказують на послідовність дій. Лінії можуть закінчуватись стрілками, але зазвичай стрілки не ставлять, якщо лінії відображають природню послідовність дій. В іншому разі стрілки обов'язкові. Природня послідовність: зверху вниз та зліва направо. Розглянемо найбільш вживані блоки.

Таблиця – Основні графічні компоненти блок-схем

Найменування	Позначення	Функції
Початок-завершення		Початок, кінець, переривання обробки даних
Введення-виведення		Перетворення даних у форму, придатну для обробки (введення) або відображення результатів обробки (виведення)
Процес		Виконання операції або групи операцій, в результаті дії яких змінюється значення, форма представлення або розташування даних

Розгалуження		Вибір напряму виконання алгоритму залежно від деяких змінних умов
Зумовлений процес		Використання раніше створених або окремо написаних програм (підпрограм)

Залежності від дій, які потрібно виконати, алгоритми поділяють на такі класи: лінійні, розгалужені, циклічні.

Лінійні алгоритми. Команди виконуються у природньому порядку, оскільки вони записані зверху вниз.

Розгалужені алгоритми. В цих алгоритмах залежно від деяких умов виконуються ті чи інші дії.

Циклічні алгоритми. Якщо певні дії необхідно повторювати декілька раз, то використовують спеціальну конструкцію – цикл. Команди циклу містять умову для визначення кількості повторень.

Блок-схеми таких алгоритмів в загальному випадку виглядатимуть так:

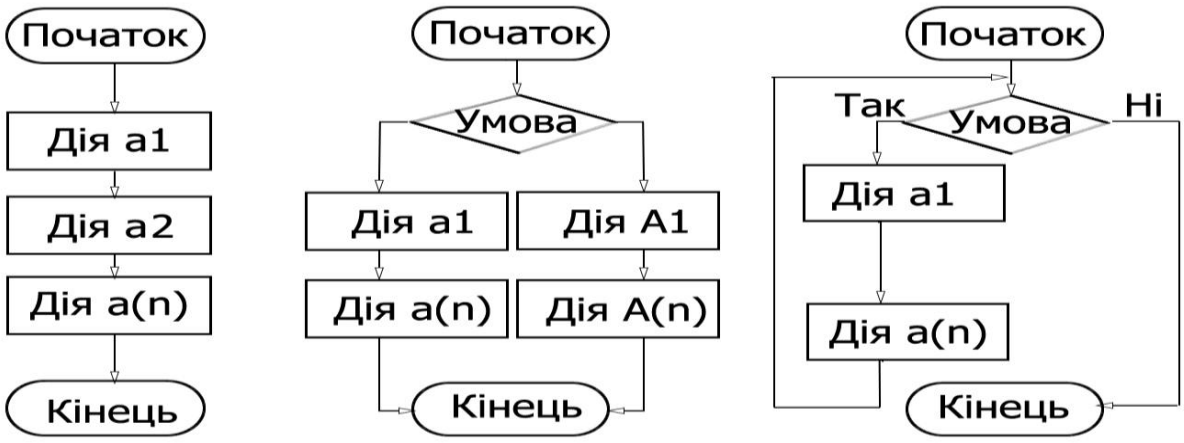


Рисунок 1.1 – Блок-схеми лінійного, розгалуженого, циклічного алгоритмів відповідно

Перевага представлення алгоритмів у вигляді схем програм – наочність. Недолік –трудомісткість виконання.

Ще одним способом запису алгоритмів є **псевдокоди**. Вони займають середню позицію між природньою і формальною мовами. Єдиного стандарту для псевдокоду не існує. Псевдокод використовує деякі службові слова (вони підкреслюються), але в той же час дозволяє твердження і на природній мові. Наведемо приклади службових слів псевдокоду:

```
для і від 1 до 10 вик.  вказівка на серію дій
якщо a > 0              перевірка умови
     то c = 0
інакше c = 1
кінець якщо
```

```
поки i < 10              команда повторення
     i = i + 1
друк i
```

Рисунок 1.2 – Запис псевдокоду

Тестові завдання

1. Який етап розв'язання задачі на комп'ютері передує написанню безпосередньо коду?

А. Запуск програми.

Б. Аналіз задачі та розробка алгоритму.

В.Тестування та налагодження.

2. Яке двійкове число відповідає десятковому числу 6_{10} ?

А. 110_2.

Б. 011_2.

В. 101_2.

3. Що таке алгоритм?

А. Синтаксис конкретної мови програмування.

Б. Скінченна, чітко визначена послідовність дій для розв'язання певної задачі.

В. Набір змінних, що використовуються в програмі.

4. До якої системи числення найлегше перевести двійкове число, групуючи його розряди по три?

А. Вісімкова.

Б. Десяткова.

В. Шістнадцяткова.

5. Яка парадигма програмування фокусується на створенні об’єктів, що поєднують у собі дані та функції для їх обробки?

А. Об’єктно-орієнтована.

Б. Функціональна.

В. Процедурна.

6. Що є основою для перетворення цілого числа з десяткової системи в будь-яку іншу позиційну систему числення (з основою **N**)?

А. Використання степенів 10.

Б. Послідовне ділення на основу N із записом остач.

В. Послідовне множення на основу N.

7. Яке десяткове число відповідає шістнадцятьковому числу 5_{16} ?

А. 80

Б. 16

В. 5

8. У якому порядку зазвичай виконуються етапи розв'язання задачі?

А. Алгоритмізація → Аналіз → Тестування.

Б. Кодування → Алгоритмізація → Тестування.

В. Аналіз → Алгоритмізація → Кодування → Тестування.

9. Перетворення якого двійкового числа у вісімкове дає 12_8 ?

А. 110_2.

Б. 001010_2.

В. 010100_2.

10. В якій парадигмі програмування основний фокус зроблений на послідовності команд (процедур/функцій) для зміни стану програми?

А. Імперативна (процедурна).

Б. Об’єктно-орієнтована.

В. Функціональна.

Практичні завдання

1. Розробіть алгоритм (можна у вигляді блок-схеми або псевдокоду) для перетворення 8-бітового двійкового числа у шістнадцяткову та вісімкову системи числення, використовуючи **метод групування** (по 4 біти для Hex, по 3 біти для Octal).

2. Проаналізуйте наведений нижче фрагмент коду та визначте, до якої основної парадигми програмування він належить.

```
// Логіка для обчислення площі прямокутника
  struct Data {
      double width;
      double height;
  };

  // Функція, яка приймає структуру та обчислює площу
  double calculate_area(struct Data rect) {
      return rect.width * rect.height;
  }
```

Контрольні запитання

1. Який з етапів розв’язання задачі за допомогою комп’ютера є найбільш критичним для забезпечення коректності та ефективності програми?

2. Чому перетворення з двійкової системи числення у шістнадцяткову не вимагає проміжного переведення в десяткову систему?

3. Яке математичне правило лежить в основі методу перетворення числа з будь-якої позиційної системи числення у десіткову систему?

4. Поясніть, що таке скінченність алгоритму? Чому ця властивість є критичною для комп’ютерних програм?

5. У чому полягає перевага двійкової системи числення у контексті представлення даних у пам'яті?

Тема 2. Елементи мови програмування С

План

1. Елементи мови С.
2. Структура С-програми.
3. Типи даних.
4. Змінні.
5. Базові операції над даними.
6. Перетворення типів.
7. Форматоване виведення даних.
8. Форматоване введення даних
9. Програми з лінійною структурою

2.1 Елементи мови С

Історія мови С тісно пов'язана з появою операційної системи Unix, яку було розроблено 1969 р. у фірмі Bell Labs для міні-ЕОМ. На початку 60-х рр. створено мову програмування Алгол 60, потім у Кембриджі 1963 і 1967 рр.- мови CPL (Combined Programming Language) та BCPL (Basic CPL). Компілятор для мови програмування В розробив Кен Томпсон 1970р. у фірмі Bell Labs, мову програмування С - Денніс Рітчі 1972 р. також у фірмі Bell Labs. Далі розвиток відбувався у напрямку створення компіляторів для удосконалених мов програмування С++ та С#.

Алфавіт мови програмування – це скінченний набір символів. За допомогою цих символів можуть бути записані ідентифікатори, вирази та оператори мови.

- малі латинські символи, великі латинські символи, десяткові цифри від 0 до 9, символ підкреслення);
- розділовий символ пробілу;
- спеціальні символи (+ - * / = > < . , ; : ‘ ( ) [ ] { } @ # $ & | %).

Із символів алфавіту складаються лексичні одиниці мови, або **лексеми** – мінімальні значущі одиниці в текстах програм. Множина всіх допустимих лексем називається **словником мови програмування**.

Лексеми спеціальних символів, окрім спеціальних символів з алфавіту мови, містять ще складені спеціальні символи, що сприймаються компілятором як єдине ціле (<=, >=, +=, *=, ++, ==, -- і т.д.).

Зарезервовані (ключові) слова мають строго визначений зміст. Їх призначення не може змінюватися. Зарезервовані слова використовуються для позначення алгоритмічних конструкцій (циклів, умов, операторів), типів змінних тощо.

Функції мови С, складаються з операторів. Кожен оператор викликає виконання певних дій на відповідному кроці виконання програми. Оператори мови розділяються символом «;» (крапка з комою).

Для написання операторів застосовуються латинські прописні та рядкові букви, цифри, спеціальні знаки. До таких знаків, наприклад, відносяться: крапка, кома, двокрапка, крапка з комою та ін.

2.2 Структура С-програми

В загальному випадку програма мовою С має таку структуру:

1. Коментар про визначння програми.
2. Підключення бібліотек.
3. Визначення процесора (констант, макросів тощо).
4. Оголошення прототипів функції.
5. Оголошення глобальних змінних.
6. Визначення головної функції.
    6.1. Оголошення локальних змінних.
    6.2. Виклик функцій.
    6.3. Оператори.
7. Визначення інших функцій.

Перетворення вихідної програми, написаної мовою С, у код виконання (файл *.exe) відбувається у три етапи: препроцесування, компіляція і компоновка. Препроцесор підключає зовнішні файли і розширює макроси. Компілятор транслює результати роботи препроцесора в об’єктний код (файл *.obj), якщо в тексті програми немає синтаксичних і семантичних помилок. Компоновщик зв'язує об’єктний код, створений компілятором, із програмами з бібліотек або іншими файлами. Результатом компоновки є exe-файл (див. рис. 2.1).

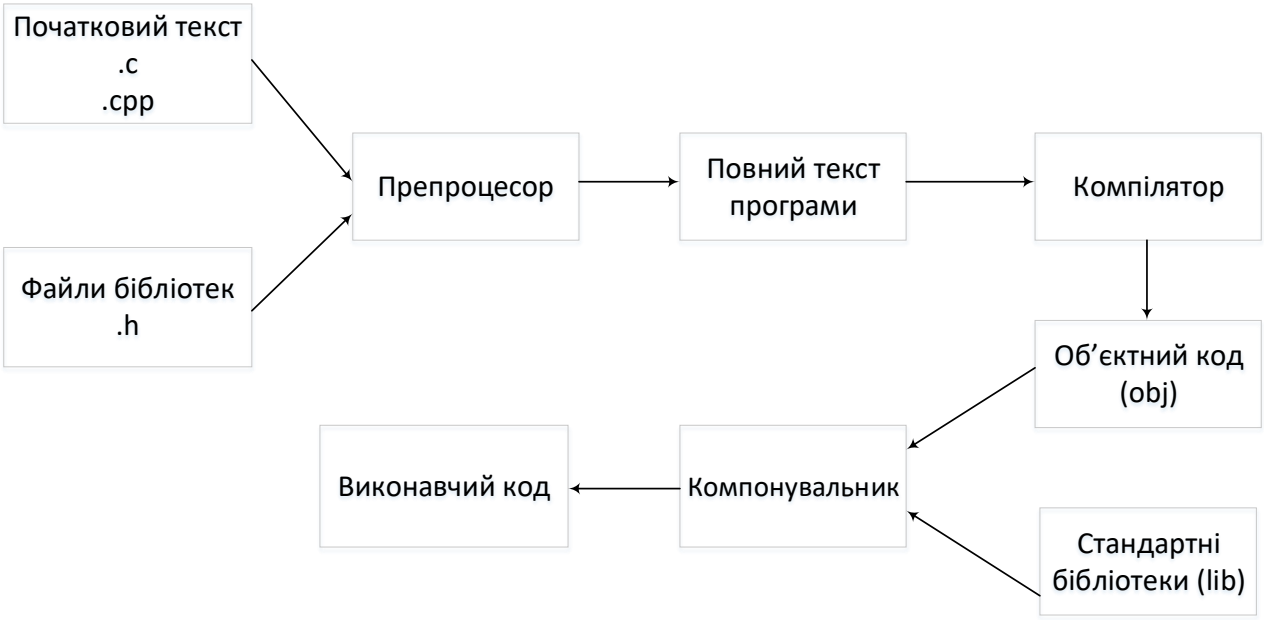


Рисунок 2.1 – Етапи перетворення програми у виконуваний код

Програма мовою С – це рядок символів, що складається з таких лексичних елементів: ключові (зарезервовані) слова, константи, операції, розділювачі та ідентифікатори.

Будь-яка програма, що створено мовою програмування С, складається як мінімум з однієї функції **main( )**, яка є головною. Тіло функції записується у фігурних дужках. В програмах на мові С важлива роль відводиться коментарям. Вони підвищують наочність та зручність читання програм. Коментарі виділяються символами **/*** и ***/**. Їх можна записувати у будь-якому місці програми. Інший спосіб виділення коментарів, це знак **//**, а все, що знаходиться після і до кінця поточного рядка, буде також розглядатися як коментар (див. рис. 2.2). Помилкою буде вважати, що за кодом програми, який не містить коментарів, можна легко зрозуміти зміст програми (навіть її розробнику).

Пробіли, символи табуляції та переходу на новий рядок у програмах на С ігноруються. Це дозволяє записувати різні вирази у візуально сприятливому вигляді. Крім того, рядки програми можна починати з будь-якої позиції, що дає можливість виділяти в тексті групи операторів.

```
#include <stdio.h> //підключення заголовного файлу
int main() /*проста програма*/
{
 cout <<"Hello world\n";} //неформатоване виведення на екран
```

Рисунок 2.2 – Приклад оформлення коментарів

#include <stdio.h> – підключення іншого файлу. Цей рядок вказує компілятору, що потрібно задіяти інформацію, що міститься у файлі **stdio.h** (ця конструкція називається **директивою**). У програмуванні термін «директива» використовується для опису певних конструкцій мови програмування. Узагальнений формат директиви препроцесора має такий вигляд:

```
# <ім'я_директиви><лексеми_препроцесора>
```

Рисунок 2.3 – Узагальнений формат директиви препроцесора

Таблиця 2.1 – Препроцесорні директиви мови С

Директива	Пояснення
#define	визначення макроса чи препроцесорного ідентифікатора
#include	підключення тексту з файла
#undef	скасування визначення макросу чи препроцесорного ідентифікатора
#if	перевірка умови-виразу
#ifdef	перевірка визначеності ідентифікатора
#ifndef	перевірка невизначеності ідентифікатора
#else	початок альтернативної гілки для **#if**

Директива	Пояснення
#endif	завершення умовної директиви **#if**
#elif	складена директива **#else/#if**
#line	ім’я файла і бажаний початковий номер рядка
#error	формування тексту повідомлення про помилку при трансляції
#pragma	дії, передбачені реалізацією
#	порожня директива

2.3 Типи даних

Типом даних визначаються набір дозволених значень і набір операцій, які можна виконувати зі змінними цього типу. Кожен тип даних (у кожній реалізації мови С) займає певну ємність пам'яті (**char** − 1 байт, **int** – 2 або 4 байт тощо). Ємність пам’яті для елементів можна обчислити за допомогою операції **sizeof**.

Перед використанням змінних цілого типу їх треба описати специфікатором відповідного типу, наприклад: **int age**; **float pi=3.14**; **char c='z'**;. Якщо застосовано специфікатор **unsigned**, **long** або **short**, то специфікатор **int** можна не наводити.

Тип даних визначає розмір і тип інформації, яку зберігатиме змінна. Нижче в таблиці наведено найосновніші типи.

Таблиця 2.2 – Основні типи даних С

Тип даних	Розмір	Опис
int	2 або 4 байти	Зберігає цілі числа без десяткових знаків
float	4 байти	Зберігає дробові числа, що містять один або більше десяткових знаків. Достатньо для збереження 6 десяткових цифр
double	8 байт	Зберігає дробові числа, що містять один або більше десяткових знаків. Достатньо для зберігання 15 десяткових цифр
char	1 байт	Зберігає один символ/букву/цифру або значення ASCII
bool	1байт	Логічний тип даних
enum	2 або 4	Перелік
void		Порожній, без значення, Використовується при роботі з функціями, вказівниками

У мові С для відокремлення цілої частини від дійсної використовується десяткова крапка. Крім звичайної форми, дійсні константи можна записувати у формі з плаваючою крапкою. Число перед знаком «е» називається мантисою, а після нього порядком, тобто замість основи 10 використовується літера «е», після якої вказується показник степеня (див. рис. 2.4).

$6.34e3 \Rightarrow$ позначає $6.34 * 10^3$
$4.36e-3 \Rightarrow$ позначає $4.36 * 10^{-3}$
$6e3 \Rightarrow$ позначає $6 * 10^3$
$-2.4e-8 \Rightarrow$ позначає $-2.4 * 10^{-8}$

Рисунок 2.4 – Представлення дійсних чисел в мові С

Змінна логічного типу **`bool`** може отримувати значення **`true`** (істина) та **`false`** (хиба). Також змінним логічного типу можна присвоювати цілочисельні значення. Так, 0 (ноль) сприйматиметься як **`false`**, а будь-яке ненульове значення – як **`true`**.

2.4 Змінні

Змінні – це перший і найважливіший крок до розуміння програмування. Вони є будівельними блоками, які дозволяють програмі зберігати та маніпулювати інформацією.

Змінна – це ділянка оперативної пам'яті, яка має власне ім'я і використовується для зберігання даних під час роботи програми.

У мові С кожна змінна характеризується трьома основними елементами:

1. **Ім'я** (Ідентифікатор) – це ім'я, значення якого може варіюватися від програми до програми або навіть у межах однієї програми (наприклад, **`age`**, **`total_sum`**).

2. **Тип даних –** визначає, який тип інформації може зберігатися (наприклад, ціле число, число з плаваючою комою, символ) і скільки байтів пам'яті буде виділено для цього контейнера.

3. **Значення змінної** – Фактичні дані, що зберігаються у цьому місці пам'яті.

У мові С\С++ розрізняють стандартні ідентифікатори та ідентифікатори користувача.

Стандартні ідентифікатори: імена вбудованих у мову функцій (**`cin`**, **`sin`**, **`cos`** тощо), типів даних (**`int`**, **`float`**, **`char`** тощо) і специфікаторів (**`inline`**, **`static`**, **`virtual`** тощо).

Правила вибору ідентифікаторів:

1. Перший символ – літера латинського алфавіту (**`a`**,...,**`z`**, **`A`**,...,**`Z`**) або символа підкреслювання (**_**).
2. Допустимі символи: літери англійського алфавіту, символ підкреслювання, цифри (0,...,9).
3. Мова С чутлива до регістра. Це означає, що такі ідентифікатори вважаються різними, наприклад: **`name`**, **`NaMe`**, **`NAME`**.
4. Ідентифікатори можуть бути різної довжини, але сприймається і використовується для розрізнення об'єктів (функцій, змінних, констант та ін.) тільки частина символів.

Їхня кількість змінюється для різних систем програмування, але у відповідності до стандарту ANSI C не перевищує 32 (у C++ це обмеження знято). Якщо довжину ідентифікатора встановлено рівною 5, то імена **count** и **counter** будуть ідентичними, оскільки в них співпадають перші п'ять символів.

5. Ідентифікатори для нових об'єктів не повинні співпадати з ключовими словами мови та іменами стандартних функцій з бібліотеки.

Використання раціональних ідентифікаторів змінних та імен функцій полегшує коментування і читання програмного коду (наприклад, доцільно вживати змістовні назви: **int iFlag**, **CubeSum()**. Зарезервованими словами (такими як **break**, **char**, **const**, **do**, **float**, **return**, **sizeof** тощо) не можна користуватися як ідентифікаторами (див. Додаток А).

Загалом, основні правила іменування, оформлення коду програм описано в C++ Programming Style Guidelines.

Константи бувають чотирьох типів: *цілі*, *дійсні*, *символьні* і *рядкові*. **Константи цілого типу** – це числа у вісімковій, десятковій або шістнадцятковій системах числення:

- **десяткові константи** складаються з цифр, причому першою не має бути нуль;

- **вісімкові** починаються з нуля і містять цифри від 0 до 7;

- **шістнадцяткові** починаються з нуля, потім слідує **x** або **X**, а далі наводяться шістнадцяткові символи (**0..9**, **A..F**).

Символьні константи – це символи, обмежені одинарними лапками. Такі константи мають тип **int** і збігаються з кодами символів **ASCII**. Рекомендується користуватися символьними константами, а не їх цифровими кодами (наприклад, **ch = 'a'** краще, ніж **ch=97**). Керуючі коди (ескейппослідовності) також беруть у лапки, наприклад: **'\n'** – символ переведення рядка, **'\t'** – горизонтальна табуляція. **'\ddd'** – вісімковий код символу **ASCII**. Рядкові константи беруть у подвійні лапки.

Всі **змінні** до їх використання повинні бути **визначені** (оголошені). У мові С змінні можуть бути оголошені у таких місцях програми:

1. Всередині функції або блоку.
2. Поза всіма функціями
3. У визначенні параметрів функції.

Початковими значеннями змінних можуть бути як певні значення, так і вирази. Оголошення змінних має такий формат:

```
<тип_даних> <ім'я_змінної_1>=<значення змінної_1>
```

Рисунок 2.5 – Формат оголошення змінних

Змінні в мові С можуть бути ініціалізовано при їх визначенні, наприклад:

```
int a = 25, h = 6;
```

```
char g = 'Q', k = 'm';
float r = 1.89;
long double n = r*123;
```

Рисунок 2.6 – Оголошення та ініціалізація змінних в С

2.5 Базові операції над даними

Операції над даними типу `int`:

– арифметичні: +, -,*, /, % (остача від ділення), ++ (інкремент), -- (декремент);

– логічні: && (AND), || (OR),! (NOT), = = (порівняння);

– бітові: & (AND). I (OR), ^ (XOR), - (NOT), » (правий зсув), « (лівий зсув).

Змінні дійсного типу мають таку форму запису: **`float sum = 17.85`**. Пам'ять для дійсних змінних у 2 (або 4) рази більша, ніж для даних типу **`int`**. Тому відповідні арифметичні операції виконуються повільніше. Набір операцій той самий, що й для даних типу **`int`** (за винятком бітових операцій).

Символьні змінні – це змінні типу **`char`**, які займають 1 байт. За замовчанням символьні змінні мають знак. Беззнакові символьні змінні записують таким чином: **`unsigned char`**, і вони обмежені діапазоном 0-255. Оскільки тип даних **`char`** – це цілий тип, то ним можна користуватися також разом з типом даних **`int`**, наприклад: **`int a = 3 + 'z'`**.

Порядок виконання операцій визначається порядком круглих дужок. Якщо запис не містить дужок, то операції виконуються зліва направо з урахуванням їхніх пріоритетів.

Таблиця 2.3 – Пріоритети операцій

Пріоритет	Операція
1	() []
2	! ~ + - ++ -- & * (тип) sizeof new delete тип( )
3	.* ->*
4	* / % (бінарні типу множення)
5	+ - (бінарні типу додавання)
6	<< >>
7	< <= => >
8	== !=
9	&
10	^
11	|
12	&&
13	||
14	?: (умовна або тернарна операція)
15	= *= /= %= += -= &= ^= |= <<= >>=
16	, (кома)

2.6 Перетворення типів

У мові С обчислення часто вимагають змішування різних типів даних. Наприклад, можна додати ціле число (**int**) до числа з плаваючою комою (**float**), або присвоїти значення типу **double** змінній типу **int**.

Узгодження (або перетворення) типів – це процес, у якому значення одного типу даних автоматично чи примусово змінюється на інший тип, щоб програма могла коректно виконати операцію.

Це має вирішальне значення, оскільки різні типи даних по-різному зберігаються в пам'яті (займають різну кількість байтів та мають різне внутрішнє представлення). Якщо компілятор не узгодить ці типи, результат може бути непередбачуваним або невірним.

У С розрізняють два основні механізми перетворення:

1. **Неявне**: Автоматичне перетворення, яке виконує компілятор, якщо це безпечно. Воно, як правило, відбувається від "вузького" типу до «широкого» (наприклад, **int** перетворюється на **float**).
2. **Явне**: Перетворення, яке програміст **примусово** вказує за допомогою оператора приведення. Це часто потрібно, коли є ризик втрати даних.

Розуміння правил узгодження типів дозволяє програмістам писати код, який не тільки працює, але й правильно обробляє дані без неочікуваних помилок.

Приклад. Задано змінні таких типів:

```
int a = 2;   ціле
float b = 3.6;   дійсне
int c;   ціле
//  Неявне зведення
        c = a * b;   // 2 * 3.6 = 7.2
        c = 7; //таке число буде записано в пам’ять комп’ютера
// Явне зведення
        c=(int) (b*a) //  3.6 * 2 = 7.2 = 7
```

2.7 Форматоване виведення даних

Функція **printf()** (прототип міститься у файлі **stdio.h**) забезпечує форматоване виведення. Її можна записати в такому формальному вигляді:

```
printf («керуючий рядок», аргумент _1, аргумент _2, ...);
```

Рисунок 2.7 – Формальний запис функції **printf()**

Для того, щоб вивести на екран значення змінних у рядках в місцях, де потрібно підставити значення змінної розміщують специфікатор форматування

(на рисунку 2.8 специфікатори, що відповідні змінним позначено відповідними кольорами):

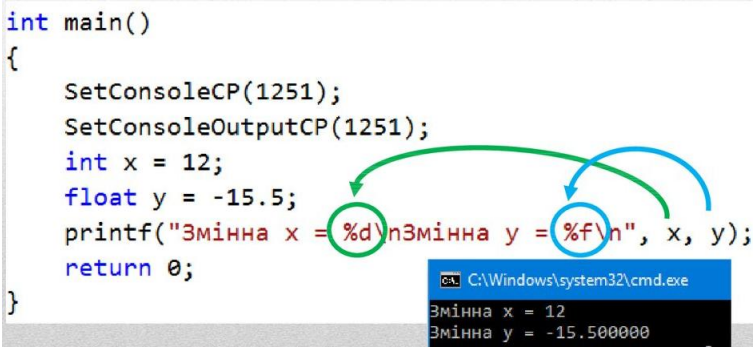


Рисунок 2.8 – Лістинг програми. Друк значень змінних

Кожна специфікація перетворення починається зі знаку `%` та закінчується деяким символом, який задає перетворення. Між знаком `%` і символом перетворення можуть зустрічатися й інші знаки у відповідності до такого формату (див. рис. 2.9):

% [ознаки] [ширина_поля][.точність] [F|N|h|l|L] c_n

Рисунок 2.9 – Загальний вигляд специфікації формату

Всі параметри в квадратних дужках не є обов'язковими:

ширина_поля – встановлює мінімальну ширину поля в результуючому рядку. Якщо результат перетворення коротший за ширину поля, відбувається вирівнювання вправо (або вліво, якщо був записаний специфікатор «-») із заповненням зайвих позицій пробілами.

.точність – специфікатор точності, що визначає кількість символів, що виводяться (залежно від прийнятого формату). Специфікатор записується після символу точки (.), щоб відокремити його від попереднього специфікатора **ширина_поля**.

Значення **c_n (символ перетворення)** наведено в таблиці нижче:

Таблиця 2.4 – Специфікатори формату C

Символ	Тип аргументу
`%d` або **`%i`**	`int`
`%f`	`float`
`%lf`	`double`
`%c`	`char`
`%s`	Використовується для рядків (тексту)

Якщо після знаку `%` записано не символ перетворення, то він виводиться на екран. Таким чином рядок `%%` призводить до виведення на екран знаку `%`.

Функція **`printf( )`** використовує керуючий рядок, щоб визначити, скільки всього аргументів і які вони мають типи. Аргументами можуть бути

змінні, константи, вирази, виклики функцій; найголовніше, щоб їхні значення відповідали заданій сертифікації (див. рис. 2.10).

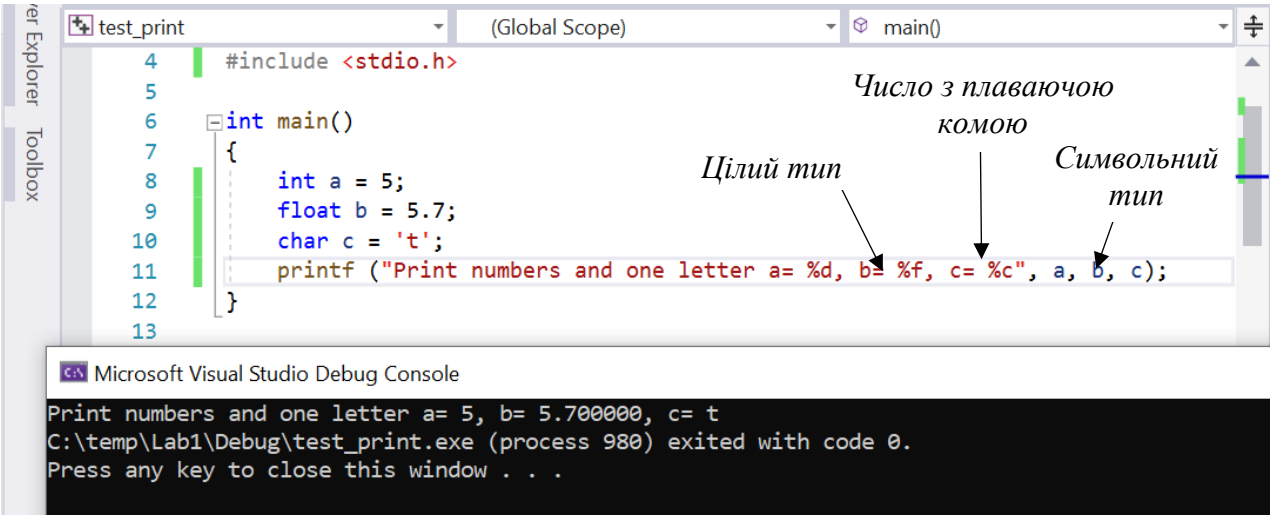


Рисунок 2.10 – Виведення змінних типів ціле, дійсне, символьне

Основні правила роботи з функцією `printf( )`:

1. Кількість специфікаторів формату має співпадати з кількістю даних для введення
2. Специфікатори формату мають точно відповідати типам даних, що виводяться

Таблиця 2.5 – Керуючі символьні константи

Символ	Опис дії
`\a`	короткочасна подача звукового сигналу
`\b`	переведення курсору вліво на одну позицію
`\f`	для подання формату
`\n`	переведення курсору на новий рядок
`\r`	для повернення каретки
`\t`	горизонтальна табуляція
`\v`	вертикальна табуляція
`\\`	виведення символу \
`\'`	апостроф
`\"`	лапки
`\?`	виведення символу ?

Приклад. В результаті виклику функції: **`printf("\tComputer\n%d\n", i);`** спочатку виконується горизонтальна табуляція (**`\t`**), тобто курсор зміститься від краю екрану, далі на екран буде виведено слово **`Computer`**, після цього курсор зміститься на початок наступного рядка (**`\n`**), далі буде надруковано ціле число **`i`** за форматом **`%d`** (десяткове ціле), далі курсор зміститься на початок нового рядка (**`\n`**).

Приклад. В результаті виклику функції: **`printf("%5.3f", number);`** буде надруковано значення змінної **`number`** з трьома цифрами після коми.

2.8 Форматоване введення даних

Програми на С, щоб бути корисними, повинні вміти взаємодіяти з користувачем, отримуючи від нього інформацію. Процес **форматованого введення даних** – це механізм, який дозволяє програмі зчитувати послідовність символів зі **стандартного входу** (зазвичай, це клавіатура) та **інтерпретувати** їх як певний тип даних, який потім зберігається у змінній.

У мові С для цього використовується функція **`scanf( )`** (прототип міститься в заголовномй файлі **`stdio.h`**). Вона потребує двох ключових елементів для успішної роботи:

1. **Рядок формату**: вказує, **який тип даних** ми очікуємо ввести (наприклад, **`%d`** для цілого числа, **`%f`** для числа з плаваючою комою).

2. **Список** адрес: вказує, куди саме у пам'яті потрібно зберегти зчитане значення (тобто, адресу змінної, використовуючи оператор **`&`**).

Її можна записати у такому формальному виді:

```
scanf("%специфікатор", &аргумент_1, &аргумент_2,...);
```

Рисунок 2.11 – Формальний запис функції **`scanf()`**

Аргументи **`scanf( )`** мають бути покажчиками на відповідні значення. Для цього перед іменем змінної записується символ **&**. Призначення покажчиків буде розглянуто далі.

Ввести ціле число (**`int a`**;), символ (**`char b`**;) та дійсне число (**`float t`**;) можна так:

```
scanf("%d", &a);
scanf("%c", &b);
scanf("%d%c%f",&a, &b, &t);
```

Рисунок 2.12 – Запис функції **`scanf( )`** з відповідними аргументами

Будь-яка невідповідність між очікуваним форматом і фактично введеними даними може призвести до несподіваної поведінки або навіть до збою програми. Правильне використання **`scanf()`** гарантує, що дані, які потрапляють у змінні, є саме того типу і формату, який очікується.

2.7 Програми з лінійною структурою

У попередньому темі було розглянуто типи алгоритмів. В цьому пукнті наведемо приклад програмної реалізації задачі з лінійною структурою.

Програма з лінійною структурою – це фундаментальний тип алгоритму, в якому всі інструкції виконуються послідовно, **одна за одною**, у тому порядку, в якому вони записані в коді. Це означає, що програма починає виконання з першої інструкції у функції **`main()`** і рухається **згори донизу**, аж до останньої інструкції.

Розглянемо приклад, в якому використовуватиметься введення даних з клавіатури. Для цього буде використано стандартну функцію **`scanf_s( )`**, яка дозволяє вводити інформацію з клавіатури під час виконання програми (див. рис. 2.13).

```
5   #include <stdio.h>
6
7   main()
8   {
9       int radius;
10      float lenght;
11      printf ("Enter value of radius: \n");
12      scanf_s("%d", &radius);
13      lenght = 3.1415 * 2 * radius;
14      printf("Радіус - %d\n, Довжина - %f\n", radius, lenght);
15  }
```

Рисунок 2.13 – Очислення довжини кола

В цій програмі порівняно з попередньою (див. рис. 2.10) з'явились нові деталі.

По-перше, оголошено змінні двох різних типів:

`radius` – типу ціле (**`int`**);

`length` – типу з плаваючою комою (**`float`**), що містить дробну частину.

По-друге використовується функція **`scanf_s`**() для введення з клавіатури значення радіуса кола. Перший аргумент функції **`scanf_s()`** - **`%d`** показує, що буде введено ціле десяткове число. Другий аргумент – ім'я змінної, якій буде привласнено введене значення. Символ **&** (амперсанд) перед іменем змінної **`radius`** необхідний для правильної роботи функції **`scanf_s()`**. Більш докладно використання цього символа обговорюватиметься пізніше.

В наступному рядку програми цілі числа **2** та **`radius`** умножаються на число з плаваючою комою **3.1415** і результат привласнюється змінній **`length`** типу **`float`**. Мова С дає можливість використання у виразах змінних різних типів. Для виведення результатів використовується функція **`printf()`**, яка містить 4 аргументи. Специфікатор формату **`%f`** використовується для друку значення змінної **`lenght`** типу **`float`**.

В розглянутому прикладі довжина кола обчислюється тільки для цілочисельних значень радіуса. Для того, щоб програма могла обчислювати довжину кола будь-якого радіусу, необхідно оголосити змінну **`radius`** як **`float`** або **`double`**, а в функції **`scanf_s()`** використовувати специфікатор **`%f`**

2.8 Основні заголовні файли С

У міру того, як програми на С стають складнішими, виникає потреба в організації коду та повторному використанні функцій. **Заголовний файл**, що зазвичай має розширення **.h**, виступає у ролі **контракту** або **каталогу** для компілятора.

Заголовний файл містить **оголошення** (декларації) функцій, які використовуються у програмі, а також визначення констант (макросів) та нових типів даних. Однак він **не містить** фактичної **реалізації** (коду) цих функцій – ця реалізація знаходиться в окремих файлах бібліотек.

Для підключення такого «каталогу» до файлу з програмою використовується **директива препроцесора**:

```
#include <назва\_файлу.h>
```

Заголовні файли дозволяють компілятору перевірити, чи правильно зроблено виклик функції (чи відповідають типи аргументів), навіть якщо сама функція фізично знаходиться в іншому місці. Це забезпечує **модульність**, дозволяє повторно використовувати код, і робить процес компіляції ефективнішим. Наприклад, щоб користуватися функцією **`scanf()`**, нам потрібно підключити заголовок **`<stdio.h>`**.

В таблиці 2.6 наведено перелік та вміст основних заголовних файлів мови програмування С.

Таблиця 2.6 – Основні заголовні файли С

Назва	Призначення
`conio.h`	Функції консольного введення виведення
`ctype.h`	Функції для перевірки літер
`math.h`	Математичні функції та макроси
`stdio.h`	Функції введення-виведення, типи, макроси
`stdlib.h`	Функції перетворення чисел, запиту пам’яті тощо
`string.h`	Функції для роботи з рядками символів
`time.h`	Типи і функції пов’язані з датою і числом

Тестові завдання

1. Яка функція є обов'язковою точкою входу, з якої починається виконання будь-якої програми, написаної на С?

А. start()

Б. void setup()

В. main()

2. Який тип даних у мові С використовується для зберігання одного символу (наприклад, 'A', '7', '!')?

А. float

Б. char

В. int

3. Який символ використовується в кінці більшості операторів (інструкцій) у мові С для позначення їхнього завершення?

А. Крапка з комою (;)

Б. Кома (,)

В. Двокрапка (:)

4. Який специфікатор формату використовується у функції printf() для коректного виведення цілого числа (int)?

А. %d.

Б. %f.

В. %s.

5. Який оператор використовується для присвоєння значення змінній у мові С?

А. Знак рівності (=).

Б. Знак 'плюс-рівно' (+=).

В. Подвійний знак рівності (==).

6. Який результат ви отримаєте після виконання наступного коду, якщо a та b є змінними типу int?

```
int a = 10;
int b = 3;
int result = a / b;
```

А. 3

Б. 3.3333

В. 1

7. Що відбувається, коли ціле число (int) неявно перетворюється на число з плаваючою комою (float)?

А. Відбувається втрата даних, оскільки float має менший діапазон.

Б. Відбувається підвищення типу (promotion), що зазвичай не призводить до втрати даних.

В. Це неможливо без явного перетворення.

8. Який оператор повертає остачу від цілочисельного ділення?

А. Оператор множення (*).

Б. Оператор модуля (%).

В. Оператор ділення (/).

9. Який символ необхідно використовувати у функції scanf() перед іменем змінної (наприклад, &value), щоб функція могла записати дані в пам’ять?

А. Зірочка (*).

Б. Амперсанд (&).

В. Двокрапка (:).

10. Який тип даних слід використовувати для зберігання великих цілих чисел, якщо стандартний int (зазвичай 32 біти) недостатній?

А. long long.

Б. short.

В. double.

Практичні завдання

1. Створіть програму, що демонструє різницю між типами int, float, double. З’ясуйте, як можна надрукувати граничні значення кожного типу.

2. Створіть програму для аналізу ASCII-коду введеного символу.

Контрольні запитання

1. Що таке ідентифікатор змінної?

2. Який оператор використовується для явного перетворення типу у мові C?

3. Як має бути оголошена змінна, щоб вона зберігала число з плаваючою комою (дробове число) з максимальною точністю?

4. Яка частина C-програми відповідає за підключення зовнішніх бібліотек і заголовних файлів (наприклад, <stdio.h>)?

5. Що означає символ \n у керуючому рядку функції printf()?

Тема 3. Програмування розгалужених процесів

План

1. Логічні операції в мові програмування С.
2. Оператор розгалуження **`if-else.`**
3. Тернарний оператор.
4. Оператор вибору варіантів **`switch`**.

3.1 Логічні операції в мові програмування С

Логічні операції – це інструменти, які дозволяють програмі оцінювати істинність або хибність одного чи кількох **умовних виразів**. Вони є математичною реалізацією булевої алгебри і є необхідними для реалізації керуючих структур, наприклад, таких як розгалуження (**`if`**) та цикли (**`while`**, **`for`**), про які мова піде в цій та наступних темах.

Результатом логічних операцій для значень типу **`bool`** є **`false`** або **`true`**, а для значень типу **`int`** є **`0`** або **`1`**. Наприклад, запис **`bool d = a > b`** означає, що змінну **`d`** буде проініціалізовано результатом операції, що записано зліва від знака присвоєння.

Логічна операція – це дія, яка виконується над логічними змінними, результатом якої є **true** або **false**

Логічний оператор приймає один чи два логічні операнди (які інтерпретуються як **істина** або **хибність**) і повертає єдиний логічний результат. У мові С:

- **Хибність (False)** представляється числом **0**.
- **Істина (True)** представляється будь-яким **ненульовим** значенням.

В мові програмування С наявні три логічні операції:

1. Логічна операція «І», позначається як **`&&`**;
2. Логічна операція «АБО», позначається як **`||`**;
3. Логічна операція «НЕ» або логічне заперечення, позначається як **`!`**.

Логічні операції утворюють складну (складену) умову з декількох простих (два або більше) умов. Ці операції спрощують структуру програмного коду в кілька разів. Можна обійтися і без складних умов, але тоді кількість операторів розгалуження збільшується в кілька разів. У таблиці 3.1 коротко охарактеризовано логічні операції в мові програмування С, що використовуються для побудови логічних умов.

Таблиця 3.1 – Опис логічних операцій

Операція	Позначення	Приклад	Короткий опис
І	&&	*a* == 3 **&&** *b* > 4 (***a*** дорівнює 3 І ***b***>4)	Складена умова істинна, якщо істинні обидві простих умови
АБО	||	*a*== 3 || *b* > 4	Складена умова істинна,

		(***a*** дорівнює 3 АБО ***b***>4)	якщо істинна, хоча б одна з простих умов
НЕ	!	*a* != 3 (***a*** НЕ дорівнює 3)	Умова істинна, якщо *a* не дорівнює 3

Оскільки комп'ютер оперує лише двома станами – **Істиною** (1) та **Хибністю** (0), –всі можливі варіанти вхідних даних можна подати у вигляді таблиці. Кожен рядок показує конкретну комбінацію умов і чіткий результат, який видасть програма. Нижче наведено результат опрацювання логічних операцій залежно від значень, які приймає логічна змінна.

Таблиця 3.2 – Таблиця істинності. Логічне заперечення

a	**!a**
true	false
false	true

Таблиця 3.3 – Таблиця істинності. Логічне **І**

a	b	**a && b**
false	false	**false**
false	true	**false**
true	false	**false**
true	true	**true**

Таблиця 3.4 – Таблиця істинності. Логічне **АБО**

a	b	**a||b**
false	false	**false**
false	true	**true**
true	false	**true**
true	true	**true**

Приклад. Записати таблицю істинності для логічного виразу

```
a && (!a || b)
```

Для формування таблиці істинності необхідно розглянути всі варіанти значень логічних змінних **`a`** та **`b`** (а вони можуть приймати значення лише **`true`** або **`false`**). Отже створюємо таблицю, куди вносимо всі можливі комбінації вхідних змінних (перші два стовпці). Далі додаємо стовпці, що містять чергову логічну операцію з урахуванням пріоритету. У нашому виразі спочатку виконується операція заперечення у дужках (третій стовпець), потім логічне «або» у дужках (четвертий стовпець) і наприкінці логічне «і». По-черзі для кожного стовпця обчислюємо значення виразу, таким чином останній стовпець містить значення всього виразу для визначених значень змінних.

a	b	!a	!a || b	**a && (!a || b)**

false	false	true	true	**false**
false	true	true	true	**false**
true	false	false	false	**false**
true	true	false	true	**true**

3.3 Оператор розгалуження `if`

Розгалуження – це механізм, який дохволяє програмі аналізувати дані та **приймати рішення**. Наприклад, програма підходить до точки розгалуження, перевіряє певну умову (наприклад, «чи є число додатним?»), і залежно від відповіді («Так» або «Ні») спрямовує потік виконання по одній з можливих гілок. В кожен конкретний момент часу може виконуватися лише **один** обраний шлях, тоді як інші ігноруються.

Оператор розгалуження if має скорочену та повну форми. Якщо умова логічного виразу істинна (результат дорівнює True або має ненульове значення), то виконується оператор, записаний після умови, інакше виконується перехід на наступний оператор програми.

Скорочена форма оператора виглядає так:

```
if (логічний вираз)
    оператор;
```

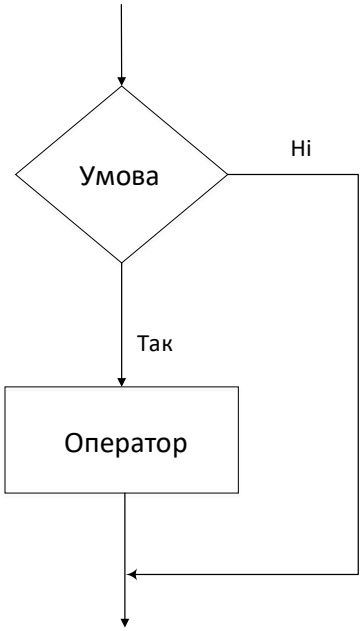


Рисунок 3.1 – Загальний вигляд та блок-схема скороченого умовного оператора

В разі використання повної форми оператора розгалуження необхідно додати альтернативний варіант роботи алгоритма. Якщо умова логічного виразу істинна (**true** або ненульове значення), то виконується *оператор 1*, інакше – виконується *оператор 2*, після чого виконується перехід на наступний оператор програми. Загальний синтаксис повної форми оператора розгалуження виглядає так:

```
if (логічний вираз) оператор 1;
else оператор 2;
```

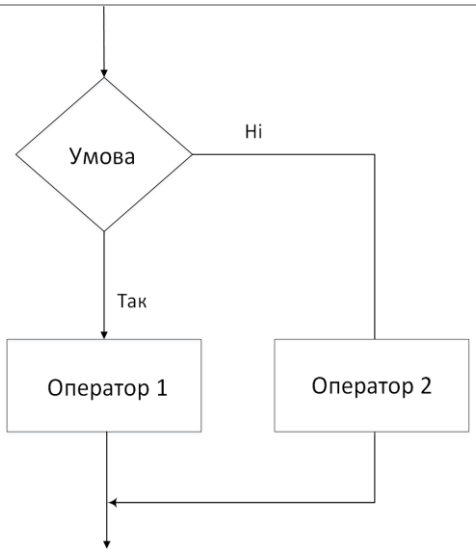


Рисунок 3.2 – Загальний вигляд та блок-схема повного оператора **if**

Конструкція **else** може бути вудсутньою, тоді за хибного результату виразу жодних дій не буде здійснено.

Приклад. Розробити програму для розв'язання квадратного рівняння із заданими користувачем коефіцієнтами.

```
#include <stdio.h>
#include <math.h>
int main(){
    double a, b, c;
    double D;
    double x_1, x_2;
    printf("Enter equation coefficients   ");
    scanf_s("%lf %lf %lf", &a, &b, &c);
    D = b * b - 4 * a * c;
    if (D >= 0){
        x_1 = ((-1) * b + sqrt(D)) / (2 * a);
        x_2 = ((-1) * b - sqrt(D)) / (2 * a);
        printf("x1 = %lf  x2 = %lf", x_1, x_2);
    }
    else
        printf("No root of equation");
    return 0;
}
```

Рисунок 3.3 – Програма обчислення коренів квадратного рівняння

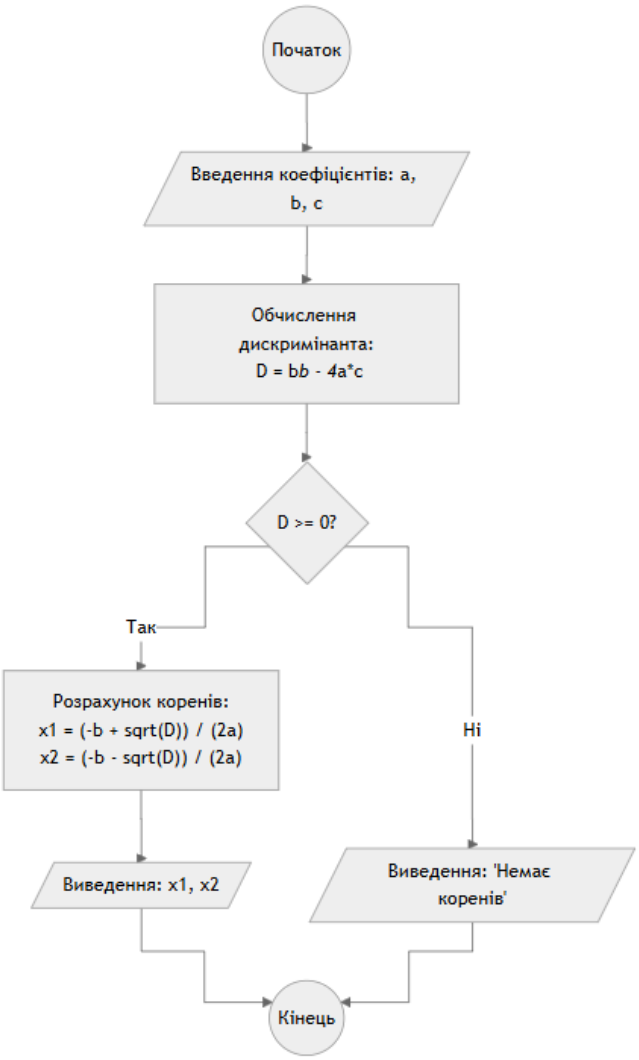


Рисунок 3.4 – Блок-схема програми обчислення коренів квадратного рівняння

Декілька коментарів до цієї програми (див. рис. 3.3):

Одразу оголошуються змінні (коефіцієнти квадратного рівняння, дискримінант, корені), визначається тип (в даному випадку **`double`**). Далі користувач вводить три числа (коефіцієнти квадратного рівняння), значення яких присвоюються змінним **`a`**, **`b`**, **`c`** відповідно. Далі можна обчислювати дискримінант за всім відомою формулою. Після проведення необхідних обчислень виконується перевірка значення дискримінанту відносно нуля. В разі успішного проходження перевірки (значення дискримінанту є більшим за нуль) – обчислюються корені квадратного рівняння. Якщо умова не набуває значення «Істина», то виконуватиметься набір операторів після конструкції **`else`**.

3.4 Тернарний оператор

Тернарний оператор (умовна операція) – це єдиний оператор у мові С, який працює одразу з **трьома** операндами. Його основна мета – спростити написання простих умовних перевірок, дозволяючи замінити конструкції **`if-else`** на лаконічний код в один рядок.

Оператор позначається символами **?** та **:** і має таку структуру:

```
умова? вираз_1 : вираз_2
```

Рисунок 3.5 – Загальний синтаксис тернарного оператора

Алгоритм виконання:

1. Обчислюється **умова**.
2. Якщо умова **істинна** (не дорівнює 0), обчислюється і повертається результат **виразу 1**.
3. Якщо умова **хибна** (дорівнює 0), обчислюється і повертається результат **виразу 2**.

Важливо: Виконується *тільки один* із виразів. Другий ігнорується.

Ключовою відмінністю тернарного оператора від конструкції **if-else** є те, що результатом його роботи завжди є **значення** (число, символ, адреса). Тому його можна використовувати всередині формул, функцій та присвоєнь. Натомість комбінація **`if-else`** – це інструкція керування**.** Вона керує потоком виконання програми, але сама по собі не має числового значення. Неможна написати **`int x = if(...)`**.

Далі наведемо декілька прикладів використання тернарного оператора та порівняємо з конструкцією **`if-else`** .

Приклад Знайти більше з двох чисел **`a`** і **`b`**.

Варіант з **`if-else`**	Варіант з тернарним оператором

`int max;`
`if (a > b) {`
`  max = a;`
`}`
`else {`
`  max = b;`
`}`
	`int max = (a > b) ? a : b;`

Варто зазначити, що тернарний оператор – це інструмент для **простих** виборів. Таку конструкцію варто використовувати для простого присвоєння, при виборі між двома значеннями. І не варто використовувати вкладені тернарні оператори (один в одному) або виконання складних дій у гілках. Прикладом такого поганого стилю (або «Спагетті-код» – один з антипатернів програмування) є подібна конструкція:

```
int result = (x > 0) ? ((y > 0) ? 1 : 2) : ((z > 0) ? 3 : 4);
```

Такий вираз важко прочитати, а тому в подібних випадках варто використовувати конструкцію **`if-else.`**

3.5 Оператор вибору варіантів `switch`

Якщо в програмі потрібно обрати один варіант із двох або трьох, конструкція **if-else** працює чудово. Втім, у випадку вибору з, наприклад 10 варіантів, можуть виникнути певні труднощі. Написання гілок **else if... else if...** зробить код громіздким і важким для читання.

В такому разі варто використовувати оператор **switch**. Це інструмент **множинного вибору**, який дозволяє порівнювати значення однієї змінної зі списком фіксованих значень (констант) і виконувати відповідний блок коду.

Загальний синтаксис оператора вибору варіантів:

```
switch (зміна вибору) {
        case const 1:   оператори 1; break;
        …
        case const N:   оператори N; break;
        default:    оператори N+1;
}
```

Рисунок 3.6 – Загальний синтаксис оператора **switch**

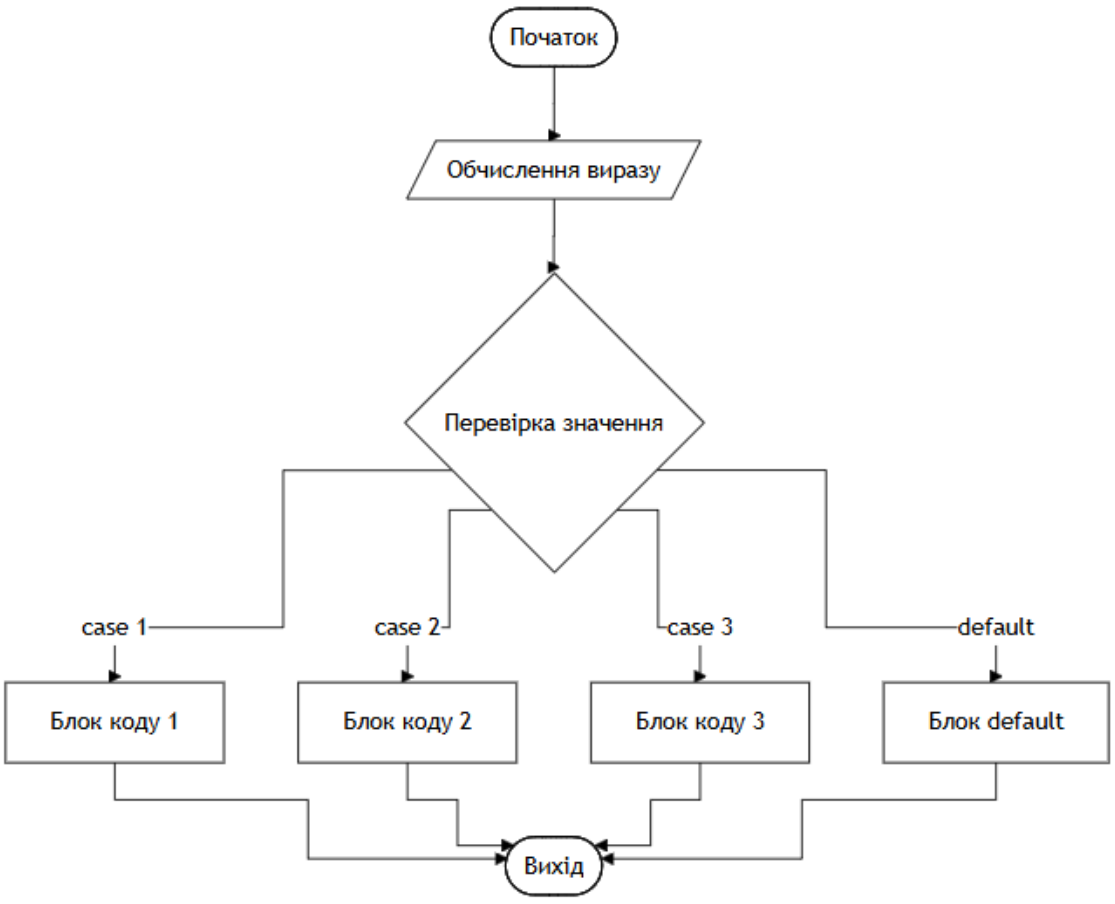


Рисунок 3.7 – Блок-схема оператора **switch**

Оператор вибору варіантів **switch** має такий формат:

1. Обчислити вираз (цілий або символьний тип) після слова **switch**. У мові С цей вираз повинен мати **цілочисельний тип** (**int**, **char**, **short**, **long**). Використання чисел з плаваючою комою (float, double) або рядків

заборонено. Результат виразу послідовно порівнюється з кожним із значень, що розташовується після ключових слів **case.**

2. Якщо значення у **switch** збігається з константою, то виконується послідовність операторів цієї гілки, що записана після двокрапки, поки не зустрінеться оператор **break**, який здійснює вихід із оператора **switch.**

3. Якщо ж значення виразу не співпало з жодним із значень гілок **case**, то виконуються оператори, записані після ключового слова **default**. Гілка **default** є необов'язковою конструкцією оператора **switch**. Якщо ж гілки **default** немає, то управління передається на наступний після конструкції **switch** оператор програми

4. Якщо після послідовності операторів певної гілки **case** відсутній оператор **break**, то будуть послідовно виконані всі оператори наступних гілок **case**, поки не зустрінеться оператор **break** або не буде досягнуто кінця **switch.**

Приклад. Розробити програму для обчислення значення виразу:

$$s = \begin{cases} |f(x)| + \ln(y), & |xy| > 10 \\ e^{f(x)+y}, & 5 < |xy| \le 10 \\ \sin(x) + \tan(y), & |xy| = 5 \end{cases}$$

При виконанні завдання запропонувати вибір виду функції ***f*(*x*)**: ***sh*(*x*)**, x^2 або e^x.

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

int main(){
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    float x, y, f, a, s;
    int key;
    //Введення даних
    printf("Введіть значення x:");
    scanf_s("%f", &x);
    printf("Введіть значення y:");
    scanf_s("%f", &y);
    printf("Оберіть вид функції f(x): 1 - sh(x), 2 - x^2, 3 -
exp(x):");
    scanf_s("%i", &key);
    switch (key){
        case 1: f = sinh(x); break;
        case 2: f = pow(x, 2); break;
        case 3: f = exp(x); break;
        default: printf("Вибір не вірний"); return 1;
    }
    a = fabs(x*y);
    if (a < 5){
        printf("Результат відсутній");
```

```
        return 1;
    }
    else
        if (a > 10) s = fabs(f) + log(y);
        else
            if (a <= 10 && a > 5) s = exp(f + y);
            else s = sin(x) + tan(y);
    printf("Значення функції: %7.3f ", s);
    return 0;
}
```

Тести для самоконтролю

1. Який логічний оператор у мові С використовується для реалізації операції логічного 'І' (AND)?

А. &&

Б. |

В. &

2. У конструкції if-else, який блок коду виконується, якщо умова в if виявляється хибною?

А. Тіло блоку else (якщо він існує).

Б. Програма завершує роботу.

В. Тіло наступного оператора switch.

3. Який результат дасть логічний вираз !(5 < 3) у мові С?

А. 2

Б. 1

В. 0

4. Яка конструкція є скороченим аналогом простого оператора if-else для присвоєння значення змінній?

А. Тернарний оператор (? :).

Б. Оператор switch.

В. Оператор goto.

5. Який оператор є логічним оператором 'АБО' (OR)?

А. ||

Б. |&

В. !

6. Яка ключова перевага використання оператора switch порівняно з ланцюжком if-else if-else?

А. Він часто більш зрозумілий і потенційно ефективніший для перевірки однієї змінної на багато фіксованих значень.

Б. Він дозволяє порівнювати змінні різних типів.

В. Він обов'язково вимагає використання блоку else.

7. Для чого використовується ключове слово break усередині оператора switch?

А. Для негайного виходу з блоку switch після виконання відповідного case.

Б. Для завершення роботи всієї функції.

В. Для позначення кінця блоку default.

8. Який оператор розгалуження слід використовувати, якщо потрібно перевірити, чи знаходиться змінна у певному діапазоні (наприклад, $10 < x < 20$)?

А. Оператор switch.

Б. Тернарний оператор.

В. Оператор if з логічним оператором &&.

9. Яка частина тернарного оператора (умова ? вираз1 : вираз2) повертається, якщо умова виявляється True?

А. вираз1 (вираз після знаку питання ?).

Б. вираз2 (вираз після двокрапки :).

В. Нічого не повертається.

10. Яке значення (True або False) має логічний вираз 15 || 0 у мові С?

А. 1 (True).

Б. 0 (False).

В. 15.

Практичні завдання

1. Напишіть програму, яка запитує у користувача вік (ціле число) та наявність дозволу (1 – є, 0 – немає). Якщо вік користувача 18 років, то «Доступ дозволено»; якщо вік менше 18, то «Доступ заборонено». Подбайте про коректне виведення кирилиці в консоль.

2. Напишіть програму, яка приймає номер дня тижня (1 – Понеділок, 7 – Неділя) і класифікує його числами від 1 до 7. Після switch використайте тернарний оператор, щоб визначити та вивести: «Це робочий день» або «Це вихідний день» (вважати 6 і 7 вихідними).

Контрольні запитання

1. Які обмеження має оператор switch щодо типів даних та перевірки умов (наприклад, чи можна перевіряти діапазони значень)?

2. Який практичний вплив має принцип «короткого замикання» на швидкодію та запобігання помилкам?

3. У чому полягає головна перевага тернарного оператора порівняно з повною конструкцією if-else?

4. Що таке «провалювання» в операторі switch?

5. Які значення повертають логічні оператори, і які – порозрядні?

Тема 4. Програмування циклічних процесів

План

1. Цикли з параметром **`for()`**.
2. Оператор циклу з передумовою **`while()`**.
3. Оператор циклу з постумовою **`do-while()`**.
4. Оператори **`break`** і **`continue.`**

У практичних задачах часто виникають ситуації, коли необхідно повторити дії задану кількість разів або до досягнення певної умови. Наприклад, вивести перелік всіх користувачів, виконати обчислення до досягнення певної точності тощо.

Обчислювальний процес називається ***циклічним***, якщо він повторюється, поки виконується певна умова.

Кожне повторення циклу називається ***ітерацією***.

Змінна, яка зберігає поточний номер ітерації, називається ***лічильником ітерацій циклу*** або ***лічильником циклу***.

Блок операторів, що повторюються, називається ***тілом циклу***.

Мова програмування С має три типи циклів:

• цикл з параметром (**`for()`**);

• з передумовою (**`while()`**);

• з післяумовою (**`do-while()`**).

Далі наведемо коротку характеристику кожного з них

4.1 Цикли з параметром `for()`

В заголовку циклу з параметром явним чином задається кількість повторень тіла циклу, а параметр циклу змінюється автоматично всередині тіла циклу.

Цикли **`for`** зазвичай використовують, коли кількість повторень відома заздалегідь або коли умова продовження виконання циклу записується коротким виразом. Загальний вигляд:

```
for(ініціалізація; умова; оновлення)
{
    <тіло циклу> // виконувати ці дії
}
```

Рисунок 4.1 – Загальний синтаксис оператора **`for`**

Блок -схема такого циклу має вид:

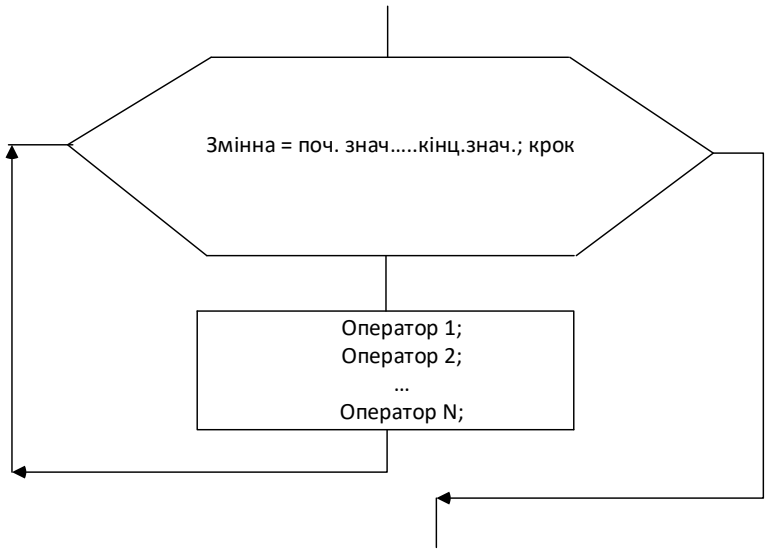


Рисунок 4.2 – Блок-схема циклу з параметром

Оператор **for** складається з трьох основних блоків, записаних у круглих дужках і відокремлених один від одного крапкою з комою, а також тіла циклу. У ***блоці ініціалізації*** задаються початкові значення змінних (параметрів), які керують циклом. У ***блоці умови*** задається та перевіряється умова і якщо вона є істинною або приймає ненульове значення, то виконується тіло циклу, інакше відбувається вихід із циклу, а керування передається на перший оператор після циклу. ***Модифікації*** змінюють параметри циклу. Перевірка умови виконується на початку кожної ітерації циклу. Якщо умова є хибною при входженні у цикл, то тіло циклу не буде виконуватись жодного разу, а керування буде передано наступному оператору програми.

В циклі **for** можуть бути або відсутні певні блоки або навпаки тіло циклу можна записати безпосередньо як параметри циклу. Цей оператор дуже гнучкий і для компактності можна використовувати і такий запис (див. рис. 4.3).

```
int F = 1, i = 1, n = 5;
   for (; i <= n;i++)
       F *= i;
```

Рисунок 4.3 – Скорочений запис оператора **for**

У разі відсутності блоку ініціалізації початкове значення змінної циклу обов'язково задається попередньо. Якщо певний блок циклу **for** відсутній, то вираз цього блока пропускається, але крапка з комою обов'язково має залишитись.

Приклад 3.1 Обчислити значення виразу (даний вираз читається так: сума чисел від 1 до 10)

$$S = \sum_{i=1}^{10} i$$

Блок-схема та програмна реалізація цієї задачі матимуть вид:

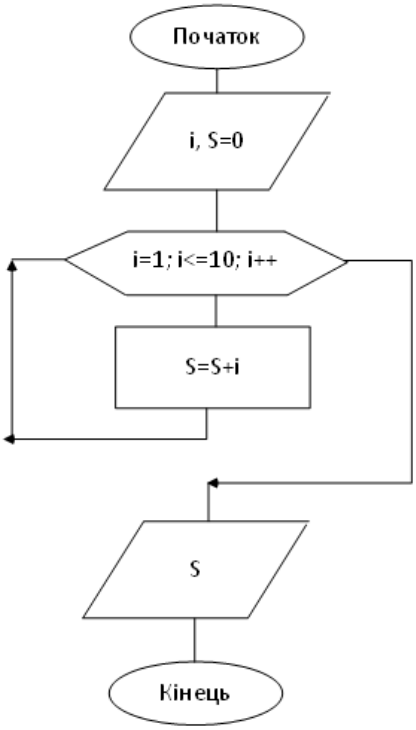


Рисунок 4.4 – Блок-схема до задачі про обчислення суми ряду

```
#include <stdio.h>

int main()
{
    int i, S = 0;
    for (i = 1; i <= 10; i++)
    {
        S += i; //аналогічно запису S = S+i
    }
    printf("Suma of numbers is equal %d", S);
}
```

Рисунок 4.5 – Лістинг програми обчислення суми

Далі в таблиці наведено поетапний процес виконання програми:

Значення лічильника	Перевірка умови	Виконання операторів в тілі цикла Значення суми	Оновлення лічильника
i = 1	1<=10 – умова істинна, виконуються оператори в тілі цикла	S = S+i=0+1 = 1	i =i+1 = 1+1 = 2
i = 2	2<=10 – умова істинна	S = S+i= 1+2 = 3	i =i+1 = 2+1 = 3

i = 3	3<=10 – умова істинна	S = S+i= 3+3 = 6	i =i+1 = 3+1 = 4
i = 4	4<=10 – умова істинна	S =S+i= 6+4 = 10	i =i+1 = 4+1 = 5
i = 5	5<=10 – умова істинна	S =S+i= 10+5 = 15	i =i+1 = 5+1 = 6
i = 6	6<=10 – умова істинна	S =S+i=15+6 = 21	i =i+1 = 6+1 = 7
i = 7	7<=10 – умова істинна	S =S+i=21+7 = 28	i =i+1 = 7+1 = 8
i = 8	8<=10 – умова істинна	S =S+i=28+8 = 36	i =i+1 = 8+1 = 9
i = 9	9<=10 – умова істинна	S =S+i=36+9 = 45	i=i+1=9+1 = 10
i = 10	10<=10 – умова істинна	S=S+i=45+10 = 55	i=i+1= 10+1 = 11
i = 11	11<=10 – умова не є істинною. Виконання циклу переривається!!!		

Змінну **i** можна ініціалізувати як до циклу, так і всередині конструкції **for**. Тобто зробити такий запис `for (int i = 1; i <= 10; i++)`

Декілька коментарів до програми. Запис **i++** означає, що змінна циклу (тобто лічильник) на кожному етапі роботи циклу змінюється з кроком 1. Окрім цього зверніть увагу на ініціалізацію нулем змінної **S**, яка накопичуватиме суму на кожній ітерації. Для чого це зроблено? На початку роботи програми, до ініціалізації, в комірці пам'яті з ім'ям **S** може зберігатись будь що.

Нижче на скріншоті у вікні налагодження програми можна побачити, які значення зберігаються в комірках **i** та **S** до ініціалізації (**-858993460**)

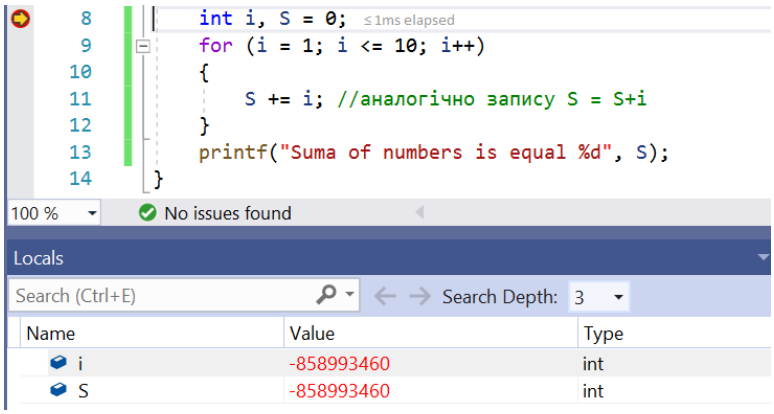
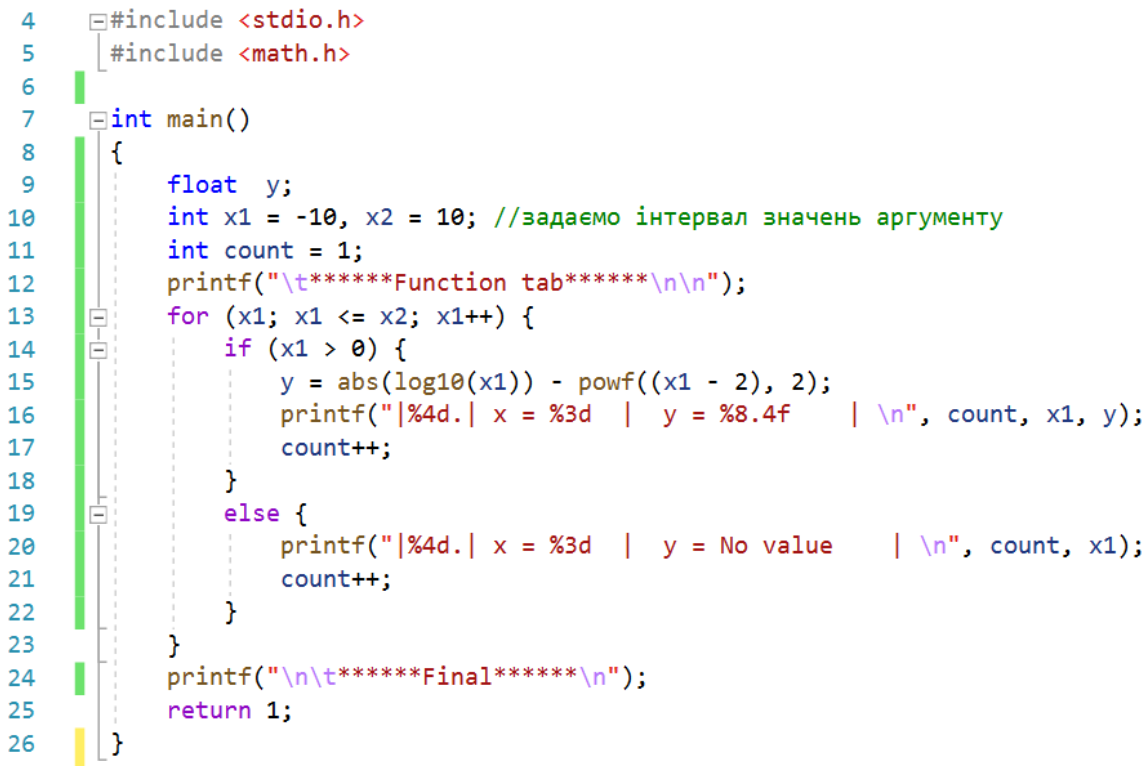


Рисунок 4.6 – Налагодження програми обчислення суми

Приклад. Протабулювати функцію $y = |\log_{10}(x)| - (x-2)^2$ на проміжку $[x1; x2]$

Програмна реалізація даної задачі наведена на рисунку 4.7. Нижче буде наведно коментарі до основних етапів роботи програми.



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

int main()
{
    float  y;
    int x1 = -10, x2 = 10; //задаємо інтервал значень аргументу
    int count = 1;
    printf("\t******Function tab******\n\n");
    for (x1; x1 <= x2; x1++) {
        if (x1 > 0) {
            y = abs(log10(x1)) - powf((x1 - 2), 2);
            printf("|%4d.| x = %3d  |  y = %8.4f    | \n", count, x1, y);
            count++;
        }
        else {
            printf("|%4d.| x = %3d  |  y = No value    | \n", count, x1);
            count++;
        }
    }
    printf("\n\t******Final******\n");
    return 1;
}
```

Рисунок 4.7 – Лістинг програми табулювання функції

Декілька коментарів до цієї програми. Інтервал значень аргументу задаємо безпосередньо в програмі. Звісно початкові і кінцеві значення аргументів можна задати з клавіатури. Певну кількість пробілів в **`printf`** додано для гарного виведення на екран. Змінну **`count`** введено для відображення номеру рядка в таблиці. В рядку № 16 параметр **`8.4`** означає, що 8 – це загальна кількість знакомісць, яке відводиться під значення змінної, 4 – кількість позицій для дробної частини (тобто після десяткової крапки). У цьому прикладі кількість знакомісць у виводимому числі менша за 8, тому вільні місця зліва від числа заповнюються пробілами. Такий спосіб форматування доцільно використовувати для побудови таблиць.

При використанні вкладених циклів потрібно слідкувати, щоб внутрішній цикл повністю вкладався у тіло зовнішнього циклу. Крім того, внутрішній цикл може містити власні вкладені цикли. Обмеження на кількість вкладених циклів немає. Імена параметрів зовнішнього та внутрішнього циклів повинні бути різними.

Прикла. Написати програму, яка друкує на екрані таблицю Піфагора

```c
#include <stdio.h>
int main()
{
    int i, j;
    printf("    ");
    for (j = 1; j <= 10; j++) {
        printf("%4d", j);
    }
    printf("\n");
```

```
        for (i = 1; i <= 10; i++) {
            printf("%4d", i);
            for (j = 1; j <= 10; j++) {
                printf("%4d", i * j);
            }
            printf("\n");
        }
        printf("\nClick ENTER to finish");
```

Рисунок 4.8 – Лістинг програми «Таблиця Піфагора»

Таким чином, в заголовку циклу явним чином задається кількість повторень тіла циклу, а параметр циклу змінюється автоматично всередині тіла циклу.

4.2 Цикл з передумовою

Цикли з передумовою (**`while()`**) переважно використовуються тоді, коли кількість ітерацій (повторень) заздалегідь невідома. Натомість, є умова, істинність якої перевіряється при кожному проході циклу:

```
        while (умова)
        {
          // тіло циклу
        }
```

Рисунок 4.9 – Загальний синтаксис циклу з передумовою

Схематично роботу такого алгоритму можна представити так:

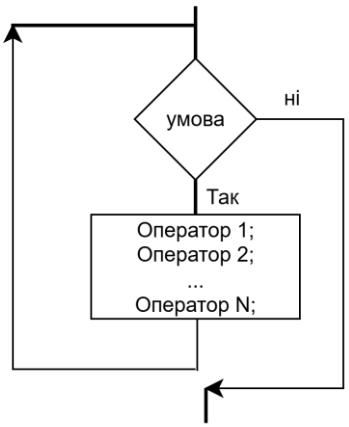


Рисунок 4.10 – Фрагмент блок-схеми циклу з передумовою

Умова циклу перевіряється перед кожною ітерацією. Тіло циклу виконується, поки умова є істинною (або має ненульове значення), а вихід з циклу здійснюється в разі хибності умови (або має нульове значення). Якщо умова є хибною при входженні в цикл, то послідовність операторів не виконається жодного разу, а керування буде передано наступному оператору програми.

Якщо є необхідність обчислити кількість ітерацій – можна ввести додаткову змінну, значення якої буде оновлюватись кожного разу після виконання тіла циклу. Таку змінну необхідно ініціалізувати до циклу.

Приклад. Вивести на екран числа від 1 до 10 в зворотньому порядку:

```
int count = 10;
while (count >= 0)
{
    printf("%i\n", count);
    count--;
}
```

Рисунок 4.11 – Програма друку чисел в зворотньому порядку

Програма виконуватиметься доти, доки змінна **count** не набуде значення -1. Таким чином, на екрані буде надруковано числа від 1 до 10. Зменшення значення змінної **count** на одиницю на кожній ітерації циклу забезпечуватиме друк в зворотньому порядку.

Отже, в циклі з передумовою спочатку перевіряється умова, і якщо вона є істинною, то наступним етапом виконується послідовність операторів в тілі циклу. Якщо умова **не виконується**, то цикл **завершується** і управління передається оператору, який записано після циклічного блоку. При невиконанні умови цикл може взагалі не виконатись жодного разу.

4.3 Цикл з післяумовою

Циклічний оператор виконує дію, потім перевіряє умову. Якщо умова істинна, то повторює дію, знову перевіряє умову і так далі:

Відмінність **циклу з післяумовою do-while** від циклу з передумовою полягає в тому, що цикл з післяумовою завжди виконується принаймні один раз незалежно від істинності чи хибності умови циклу. Загальний синтаксис оператора наведено на рисунку:

```
do
{
  // виконувати поки умова істинна
}
while (умова);
```

Рисунок 4.12 – Загальний синтаксис циклу з післяумовою

Тіло циклу з післяумовою виконується поки умова є істинна. Якщо тіло циклу складається з одного оператору, то операторні дужки необов’язкові.

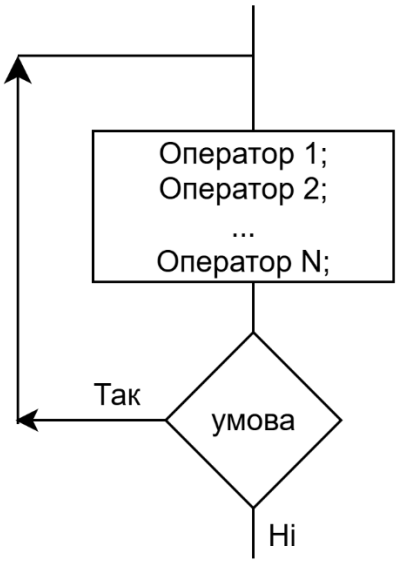


Рисунок 4.13 – Блок-схема циклу з післяумовою

У тілі циклу з післяумовою потрібно передбачити зміну параметрів, які використовуються в умові, інакше буде відбуватись зациклення.

Цикл **do-while**, на відміну від **while**, обов'язково виконується хоча б один раз, навіть якщо умова хибна. Зазвичай цикл з постумовою застосовується тоді, коли необхідно виконати цикл хоча б один раз, наприклад, при очікуванні введення від користувача.

Приклад. Програма запитує введення числа у користувача, допоки він не введе додатнє число:

```
int input;
do
{
    printf("Enter a positive number: ");
    input = GetInt();
}
while(input < 1);
```

Рисунок 4.14 – Програмна реалізація циклу з постумовою

Умова перевіряється після дій в тілі циклу. І якщо ця умова **виконується**, то відбувається вихід з циклу. Такий цикл виконається хоча б один раз.

4.4 Оператори `break i continue`

У стандартних алгоритмах цикли (**for**, **while**, **do-while**) виконуються доти, доки умова в заголовку циклу залишається істинною. Однак, іноді виникають ситуації, коли необхідно достроково зупинити цикл або пропустити поточний крок.

Для цього в мові С існують оператори безумовного переходу: **break** та **continue**.

Оператор **break** використовується для **миттєвого виходу** з циклу або конструкції **switch**. Він перериває нормальний потік виконання і передає керування першій інструкції, що знаходиться після закриваючої дужки циклу. Коли програма зустрічає **break**, вона ігнорує всі команди, що залишилися в тілі циклу, ігнорує умову продовження циклу і просто виходить з нього. Нижче наведено приклад застосування цього оператора (див. рис. 4.15)

```
int i;
for (i = 0; i < 10; i++) {
    int input;
    scanf("%d", &input);

    if (input < 0) {
        printf("Знайдено від'ємне число! Зупинка.\n");
        break; // Вихід з циклу for
    }
    printf("Число %d оброблено.\n", input);
}
// Сюди потрапить програма після break
     printf("Цикл завершено.");
```

Рисунок 4.15 – Застосування оператора **break** для переривання циклу

Оператор **continue** використовується, коли потрібно **пропустити решту команд** у поточній ітерації циклу і одразу перейти до наступної ітерації. На відміну від **break**, **continue не завершує** весь цикл. Він лише пропускає ітерацію циклу та переходить до перевірки умови та (у випадку **for**) до інкременту лічильника На рисунку наведено фрагмент програми з оператором **continue**, який використано для пропуску парних чисел при обробці певної числової послідовності.

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        continue; // Якщо число парне, пропускаємо printf і йдемо
до i++
    }
    printf("%d ", i); // Виконається тільки для непарних
}
```

Рисунок 4.16 – Застосування оператора **continue** для пропуску ітерації циклу

Важливим є те, що і **break**, і **continue** діють лише на **той цикл, у якому вони безпосередньо написані**. Якщо у програмі є вкладений цикл, то **break** у внутрішньому циклі зупинить тільки внутрішній цикл, а зовнішній продовжить роботу.

Тести для самоконтролю

1. Який тип циклу в мові С найбільш доцільно використовувати, коли кількість ітерацій відома заздалегідь?

А. while

Б. for

В. do-while

2. У циклі for(A; B; C), яка частина (A, B, чи C) відповідає за умову продовження виконання циклу?

А. B (Умова)

Б. C (Крок)

В. A (Ініціалізація)

3. Яка ключова відмінність циклу do-while від циклу while?

А. do-while завжди виконується хоча б один раз.

Б. do-while перевіряє умову на початку, а while – в кінці.

В. while може використовувати лише одну змінну-лічильник.

4. Для чого використовується оператор break у циклі?

А. Для негайного виходу з циклу, завершуючи його роботу.

Б. Для пропуску поточної ітерації та переходу до наступної.

В. Для перезапуску циклу з початковими значеннями.

5. Розгляньте код: `int i = 0; while (i < 5) { i++; }`. Скільки разів виконається тіло цього циклу?

А. 6

Б. 4

В. 5

6. Який цикл найкраще підходить для організації нескінченного циклу з метою постійного прослуховування вводу або обслуговування?

А. for

Б. Використання while (1).

В. do-while.

7. Що робить оператор continue у циклі?

А. Скидає лічильник циклу до початкового значення.

Б. Пропускає всі інструкції, що залишилися в тілі циклу, і переходить до перевірки умови наступної ітерації.

В. Повністю завершує цикл.

8. У циклі do-while, коли перевіряється умова?

А. Перед першою ітерацією.

Б. Після кожної ітерації (в кінці тіла циклу).

В. Умова перевіряється лише один раз.

9. Що станеться, якщо у циклі for пропустити секцію Умови?

А. Виникне синтаксична помилка, і програма не скомпілюється.

Б. Цикл виконається лише один раз.

В. Цикл стане нескінченним.

10. Якщо у циклі зустрічається оператор continue, до якого елементу циклу for(A; B; C) переходить управління?

А. До секції C (крок), а потім до B (умова).

Б. До інструкції після циклу.

B. До секції B (умова).

Практичні завдання

1. Напишіть програму, яка використовує цикл for для перевірки чисел від 1 до 50. Якщо число ділиться на 3 без остачі, пропустіть його (використовуйте continue) і не виводьте на екран. Якщо число досягає значення 40, негайно завершіть виконання циклу (використовуйте break), незалежно від того, чи дійшла програма до 50.

2. Напишіть програму, яка використовує цикл do-while для валідації вводу користувача. Запитайте користувача про введення числа-пароля (наприклад, 1234). Забезпечте, щоб перевірка умови відбулася мінімум один раз. Умова циклу має дозволяти повторне виконання, доки введене число не дорівнює 1234. Якщо пароль невірний, виведіть «Невірний пароль. Спробуйте ще раз». Після виходу з циклу (введення вірного пароля) виведіть «Доступ надано»

Контрольні запитання

1. Яким чином у мові C можна створити нескінченний цикл (використовуючи while або for)?

2. Що відбувається, якщо умова у while ніколи не стає хибною?

3. Які є функціональні відмінності між операторами break та continue у контексті циклів?

4. Яке призначення кожної з трьох секцій у заголовку циклу for(A; B; C)?

5. Коли використання do-while є обов’язковим або більш доцільним, ніж while?

Тема 5 Масиви

План

1. Основні поняття.
2. Одновимірні масиви
3. Генерування послідовності псевдовипадкових чисел
4. Двовимірні масиви

5.1 Основні поняття про масиви

Масив – це структурований тип даних, який використовується для опису впорядкованої сукупності фіксованої кількості елементів одного типу, які мають спільне ім'я. У пам'яті комп'ютера масив виглядає як послідовність комірок, розташованих одна за одною. Наведемо основні **характеристики масивів**:

1. **Однорідність:** У масиві можна зберігати дані тільки одного типу. Не можна зберегти в одному масиві і цілі числа (**`int`**), і дробові (**`float`**), і символи (**`char`**) одночасно.

2. **Неперервність пам'яті:** Елементи масиву завжди розміщуються в пам'яті послідовно один за одним. Це забезпечує дуже швидкий доступ до даних.

3. **Фіксований розмір:** При створенні статичного масиву одразу потрібно зарезервувати потрібну кількість байтів для збереження його елментів. Змінити цей розмір під час роботи програми неможливо (без використання динамічної пам'яті).

Для позначення елементів масиву використовуються ім'я змінної-масива та індекс.

Ім'я масива – ідентифікатор, який використовується для звернення до елементів масива

Розмірність масива – кількість елементів масива. Вимірність масиву визначається кількістю його індексів. Наприклад, елементи одновимірного масива (вектора) мають один індекс; двовимірного масива (матриці, таблиці) – два індекси, з яких перший – номер рядка, другий – номер стовпчика. Кількість індексів в масиві необмежена, тому можна створювати масиви різної вимірності.

Для оголошення масиву необхідно вказати **тип, ім'я, розмір**. Використовують такий синтаксис:

```
<тип елемента> <ім'я масиву> [<кількість елементів>]
```

На відміну від інших мов, у мові С не перевіряється вихід за межі масива, тому, щоб уникнути помилок, слід уважно вказувати розмірність масивів.

Елементи масиву в С індексуються з нуля. Операція доступу до елементу масиву здійснюється через вказування його індексу у квадратних дужках **`[ ]`**.

Нумерація (індексація) елементів у масиві починається з нуля. Якщо масив містить N елементів, то це означає, що він містить елементи з індексами 0, 1, 2, 3, …, N-1.

5.2 Одновимірні масиви

Якщо елементи масива мають **один індекс**, то масив називається **одновимірним або лінійним, або масив-вектор**.

Наприклад, в програмі оголошено масив з 5 п’яти елементів **`int marks[5]`**. Візуально структуру масива можна представити так:

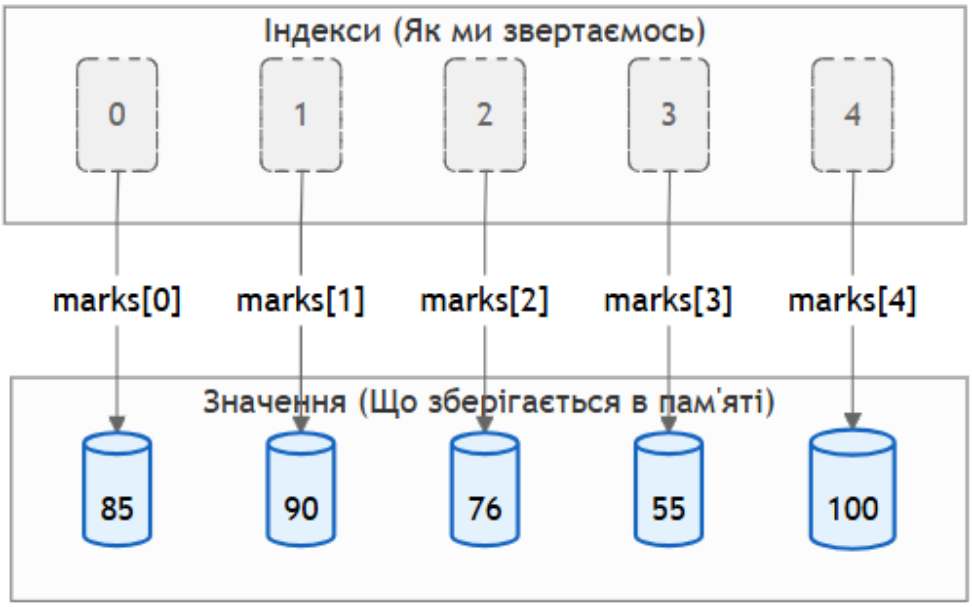


Рисунок 5.1 – Структура одновимірного масиву в пам’яті комп’ютера

Для оголошення одновимірного масиву необхідно вказати: тип значення його елементів, ім’я масиву, кількість елементів масиву (див. рис. 5.2).

```
int nums[10] = { 1, 2, 3, 4 }
int arr[] = { 1, 2, 3, 4, 5, 23, 34 }
int cards[4] = { 1, 2, 3, 4 }
```

Рисунок 5.2 – Оголошення та ініціалізація одновимірного масиву

Якщо кількість елементів у фігурних дужках менша за оголошену кількість – інші елементи масиву буде проініціалізовано нулем. Кількість елементів масиву в квадратних дужках не вказується – отже в масиві саме стільки елементів, скільки їх перераховано в фігурних дужках.

Значення елементів одновимірного масиву виводять на екран у вигляді стовпця або рядка (див. рис. 5.3).

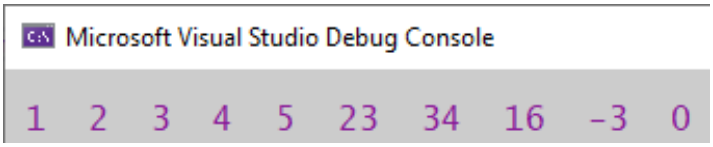


Рисунок 5.3 – Виведення одновимірного масиву на екран

Наведемо приклад обробки елементів масиву в програмі (див. рис. 5.4). Для визначення кількості елементів масиву буде використано оператор **sizeof**.

```
#include <stdio.h>

int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 23, 34, 16, -3, 0 };
    int N = sizeof arr/sizeof(int);
    for (int i = 0; i < N; i++) {
        printf(" %d ", arr[i]);
    }
}
```

Рисунок 5.4 – Приклад програми для виведення масиву на друк

В цій програмі довжина масиву **arr** в байтах обчислюється за допомогою функції **sizeof**. Для обчислення кількості елементів масиву необхідно розділити довжину масиву в байтах на кількість байт, які виділяються для типу даних **int**. Такий підхід надає можливість «не прив'язуватись» до конкретної кількості елементів масиву та динамічно змінювати дане число в залежності від умов завдання. Також елементи масиву можна ввести з клавіатури використовуючи команду **scanf()**.

Приклад. Ввести з клавіатури сім елементів масиву, обчислити середнє арифметичне елементів масиву.

```
#include <stdio.h>
int main()
{
    int const number = 7; //кількість елементів масиву
    int testArray[number]; //оголошення цілочисельного масиву
    //ініціалізація змінної для збереження суми елементів
    int sumOfArray = 0;
    //оголошення змінної для середнього значення
    double meanOfArray;
    printf("\tPlease enter the array elements\n");
    for (int i = 0; i < number; i++) {
        printf("array[%d] = ", i);
        scanf_s("%d", &testArray[i]);
    }
    printf("\n now we print our new array\n");
    for (int i = 0; i < number; i++) {
        printf("%4d", testArray[i]);
```

```
        sumOfArray = sumOfArray + testArray[i];
    }
    meanOfArray = sumOfArray / number;
    printf("\nthe mean value is: %lf", meanOfArray);}
```

Рисунок 5.5 – Обчислення середнього арифметичного елементів масиву

Зазвичай, за необхідності обробити всі або декілька (наприклад, тільки ті, які задовольняють певній умові) елементів масиву необхідно застосовувати циклічний оператор. Також в даній програмі для досягнення необхідної точночсті потрібно врахувати тип даних відповідно до якого визначається результат обчислення.

5.3 Генерування послідовності псевдовипадкових чисел

У попередніх задачах ми заповнювали масиви двома способами: в коді або послідовно вводячи елменти масивів через консоль. Це є і негнучко і повільно, зокрема, якщо мова йде про сортування елементів масиву або пошуку певного елемента при кількості елементів масиву, наприклад 100.

Саме тут **генерація псевдовипадкових чисел** стає незамінним інструментом. Вона дозволяє миттєво створити «тестовий полігон» для ваших алгоритмів. **Псевдовипадкові числа** – це послідовність чисел, яка виглядає випадковою (відсутні явні закономірності), але насправді обчислюється за складною математичною формулою.

Для роботи з випадковими числами використовується бібліотека **<stdlib.h>**. Функція **rand()** повертає ціле псевдовипадкове число в діапазоні від **0** до константи **RAND_MAX** (зазвичай це мінімум 32767). Запис в програмі **int magic_number = rand();** поверне число, наприклад, 41. Якщо запустити програму з функцією **rand()** кілька разів поспіль, то числа будуть однаковими. Чому? Тому що формула генерації за замовчуванням завжди починає обчислення з однієї й тієї ж стартової точки. Щоб послідовність чисел була різною при кожному запуску програми, нам потрібно змінити стартову точку. Для цього використовується функція **srand(unsigned int seed)**. Найпоширеніший спосіб – використати поточний час як стартову точку, оскільки час постійно змінюється. Для цього підключаємо бібліотеку **<time.h>**.

Функція **rand()** видає дуже великі числа. Щоб отримати число в потрібних межах (наприклад, від 1 до 6), використовують операцію **остачі від ділення (%)**. Таким чином, маємо таку формулу для певного діапазону **[min, max]**:

Number = min + rand() % \(max - min + 1),

де rand() % N – повертає числа від 0 до N-1, додавання **min** зміщує діапазон. Наприклад:

1. Запис **rand() % 10** повертає число від 0 до 9.
2. Запис **1 + rand() % 10** повертає число від 1 до 10.

3. Запис **`10 + rand() % 41`** повертає число від 10 до 50.

Стандартний алгоритм заповнення масиву випадковими числами виглядає так:

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

#define SIZE 10

int main() {
    srand(time(NULL)); // Запускаємо генератор
    int data[SIZE]; // резервуємо пам'ять для елементів масива
    // Заповнення масиву
    for (int i = 0; i < SIZE; i++) {
        data[i] = rand() % 100; // Числа від 0 до 99
    }
    return 0;
}
```

Рисунок 5.6 – Заповнення масиву десятьма псевдовипадковими числами

Потік даних при автоматичному заповненні масиву схематично наведено на рисунку 5.7.

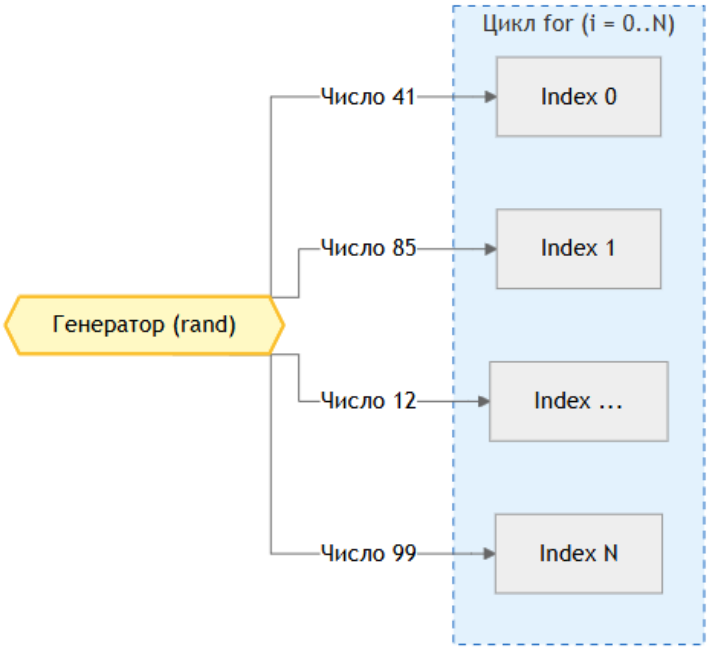


Рисунок 5.7 – Візуалізація процесу заповнення масиву псевдовипадковими числами

Це схоже на конвеєр, де генератор штампує числа, а цикл розкладає їх по комірках.

Приклад. Згенерувати масив з псевдовипадкових чисел та обчислити кількість додатніх елементів в ньому.

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

int main()
{
    int arr[20] = { 0 }; //ініціалізуємо нулем елементи масиву
    int lenghtOfArray;
    int positive; //лічильник для додатніх елементів
    printf("Enter number of elements. It may be less than 20");
    scanf_s("%d", &lenghtOfArray);
    //генеруємо випадкові числа в діапазоні від -20 до 10
    srand(time(NULL));
    for (int i = 0; i < lenghtOfArray; i++) {
        arr[i] = rand() % (10+20+1)-20;
        printf("%4d ", arr[i]);
    }
    printf("\n\n");
    //обчислимо кількість додатніх елементів
    for (int i = 0; i < lenghtOfArray; i++) {
        if (arr[i] > 0)
            positive++;
    }
    printf("\nNumber of positive elements is %d\n", positive);

    return 0;
}
```

Рисунок 5.8 – Обчислення кількості додатніх елементів масиву

На рисунку 5.8 Змінна **positive** виконує роль лічильника, який збільшується на 1 при виконанні певних умов. В даному випадку умова пов'язує збільшення лічильника на 1 з додатнім елементом, який відповідає поточному значенню лічильника циклу.

5.4 Двовимірні масиви

В деяких випадках зручно елементи масиву пронумерувати двома незалежними індексами, такі масиви називають двовимірними або матрицями.

Двовимірний масив – це масив одновимірних масивів (рядків) послідовно розташованих в пам'яті

Значення елементів таких масивів зазвичай виводять на екран у вигляді таблиці (див. рис. 5.9).

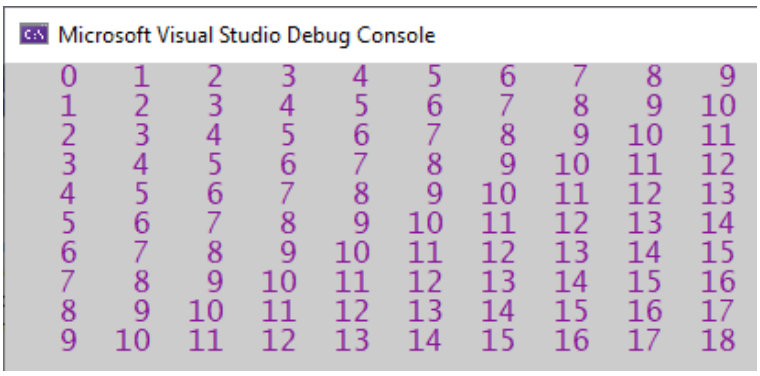


Рисунок 5.9 – Виведення двовимірного масиву на екран

Для оголошення двовимірного масиву можна використовувати такий синтаксис:

<тип елемента> <ім'яМасива> [<рядки>][<стовпці>]

Доступ до елементів здійснюватиметься за двома індексами, наприклад так **A[i][j]**, де **A** – ім'я масиву. Нижче на рисунку 5.10 наведено декілька способів оголошення та ініціалізації двовимірного масиву.

```
int a[2][3] = { 1, 2, 3, 4, 5, 6 };
int a[2][3];
int a[2][3] = { {1, 2, 3}, {4, 5, 6} };
int a[][3] = { {1, 2, 3}, {4, 5, 6} };
```

Рисунок 5.10 – Оголошення та ініціалізація двовимірного масиву

Оскільки масив двовимірний, то для його обробки необхідно використовувати конструкцію із вкладених циклічних операторів. Розглянемо відповідний приклад

Приклад. Згенерувати двовимірний масив із псевдовипадкових чисел та обчислити середнє арифметичне тих елементів, сума індексів яких є парним числом. Масив надрукувати у вигляді таблиці.

```
int main()
{
    int a[20][20] = { 0 };
    int N;
    int A, B, count = 0;
    float S = 0;
    printf("Enter interval for array generation\n");
    scanf_s("%d %d", &A, &B);
    printf("Enter size of array\n");
    scanf_s("%d", &N);
    srand(time(NULL));
    // масив заповнюємо випадковими числами з інтервалу [A; B]
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            a[i][j] = rand() % (B - A + 1) + A;
            printf("%4d", a[i][j]);
```

```
        }
        printf("\n"); //для переведення курсора на наступний рядок
і коректного друку двовимірного масиву
    }
    printf("Elements, that satisfy the condition...\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if ((i + j) % 2 == 0 ) {
                count++;
                S += a[i][j];
            }
        }
    }
    if (count != 0) {
        printf("\nNumber of elements is %d , arithmetic mean=%f",
count, S/count);
    }
    else printf("\nthere are no elements that satisfy the
condition\a");
    return 0;
     }
```

Рисунок 5.11 – Обчислення середнього арифметичного двовимірного масиву

Обробка двовимірного масиву здійснюється у вкладеній циклічній структурі, оскільки необхідно «пробігти» і по індексу рядка і по індексу стовпця. Доступ до кожного елемента здійснюється за обома індексами.

Тести для самоконтролю

1. Що таке масив у мові С?

А. Тип даних для зберігання цілого числа з великою точністю.

Б. Колекція однотипних елементів даних, що зберігаються в послідовних (суміжних) комірках пам’яті.

В. Колекція змінних різних типів, доступ до яких здійснюється за допомогою імені.

2. З якого індексу (номера) починається нумерація елементів у масиві в мові С?

А. З індексу, вказаного програмістом.

Б. 0

В. 1

3. Яка функція використовується для ініціалізації генератора псевдовипадкових чисел перед першим викликом rand()?

А. time().

Б. rand().

В. srand().

4. Як оголошується одновимірний масив цілих чисел, що містить 10 елементів?

А. int array{10};

Б. int array[10];

В. int array(10);

5. Якщо ви оголосили масив int data[5] який індекс є останнім допустимим для доступу до його елементів?

А. 5

Б. 4

В. 1

6. Що таке двовимірний масив?

А. Масив, який може зберігати лише два елементи.

Б. Масив, який використовується лише для зберігання адрес вказівників.

В. Масив, елементами якого є інші масиви (концептуально: таблиця з рядків і стовпців).

7. Яка функція в мові С використовується для генерації псевдовипадкового цілого числа?

А. random().

Б. srand().

В. rand().

8. У двовимірному масиві int matrix[5][10], що означає число 5?

А. Кількість елементів у масиві.

Б. Кількість стовпців.

В. Кількість рядків.

9. Який оператор потрібен, щоб обмежити діапазон псевдовипадкових чисел, згенерованих rand(), наприклад, до діапазону від 0 до 99?

А. Оператор додавання (+).

Б. Оператор модуля (%).

В. Оператор ділення (/).

10. Як елементи одновимірного масиву розташовуються в пам’яті комп'ютера?

А. У довільних (несуміжних) комірках пам’яті.

Б. Послідовно, безпосередньо один за одним.

В. У зворотному порядку відносно оголошення.

Практичні завдання

1. Напишіть програму, яка здійснює пошук максимального і мінімального елементів одновимірного масива.

2. Заповніть двовимірний масив псевдовипадковими числами в діапазоні від -10 до 10. Використовуючи вкладені цикли for, виведіть матрицю на екран у вигляді таблиці, щоб кожен рядок відображався на новому рядку консолі.

Контрольні запитання

1. Яке математичне правило дозволяє комп’ютеру швидко отримати доступ до будь-якого елемента масиву за його індексом (наприклад, array[i])?

2. Як оголошується та ініціалізується двовимірний масив (матриця) у C?

3. Яка пара функцій є обов’язковою для генерації різної послідовності псевдовипадкових чисел щоразу при запуску програми?

4. В чому полягає різниця між випадковими і псевдовипадковими числами?

5. Що таке вихід за межі масиву?

Тема 6. Вказівники. Створення динамічного масиву

План

1. Основні поняття. Вказівники у C/C++.
2. Операції з вказівниками.
3. Арифметика вказівників.
4. Вказівники на вказівники.
5. Вказівники на функції.
6. Динамічні масиви в C/C++.

6.1 Основні поняття. Вказівники у C/C++

Кожна змінна в програмі зберігається в оперативній пам’яті (RAM). Пам’ять – це величезна шафа з пронумерованими комірками. Кожна комірка має свій унікальний номер – **адресу** (зазвичай у шістнадцятковому форматі, наприклад **`0x7ffee4`**).

Вказівник – змінна, значенням якої є адреса комірки пам`яті. Тобто вказівник посилається на блок даних з області пам`яті, причому на саме його початок. Вказівник може посилатися на змінну або функцію. Для цього потрібно знати адресу змінної або функції. Щоб дізнатися адресу конкретної змінної в C/C++ існує унарна операція взяття адреси **&**.

Вказівники дають **прямий доступ до пам'яті**. Це дозволяє:

1. **Економити пам'ять:** Замість того, щоб копіювати величезний масив чи відеофайл при передачі у функцію, передають лише його адресу (8 байт).
2. **Змінювати оригінали:** Функції можуть змінювати змінні, які знаходяться за їх межами.
3. **Динамічна пам’ять:** Створювати структури даних, розмір яких змінюється під час роботи програми (динамічні масиви, списки, дерева).

Вказівники використовуються для передачі за посиланням даних, що набагато прискорює процес обробки цих даних (в тому випадку, якщо обсяг даних великий), через те що їх не треба копіювати, як при передачі за значенням, тобто, використовуючи ім’я змінної.

Розглянемо, як виглядає процес роботи з вказівником. Наприклад, у нас є змінна **`number`** зі значенням **`42`**. Вона знаходиться за адресою **`0x100`**. У нас є вказівник **`ptr`**, який «вказує» на цю змінну. Це означає, що всередині **`ptr`** записано число **`0x100`**.

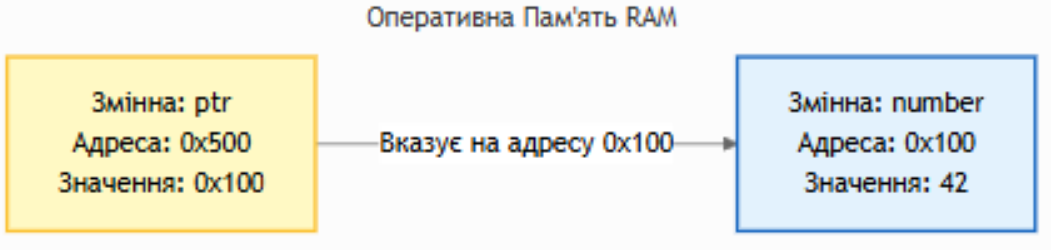


Рисунок 6.1 – Зв’язок змінної та вказівника

Будь-який вказівник необхідно оголосити перед використанням, як і будь-яку змінну (див. рис. 6.2).

```
<тип> *<ідентифікатор вказівника>
```

Рисунок 6.2 – Оголошення вказівника

Зверніть увагу на символ «*****» – це не арифметична операція множення, а обов'язковий елемент синтаксису. В програмі оголосити вказівник можна таким чином:

```
char* cptr   //вказівник на символьну змінну
int* iptr   //вказівник на цілочислову змінну
float* fptr     //вказівник на змінну дійсного типу
```

У мовах C\C++ існує вказівник, який може вказувати на будь-який тип даних. Він називається – вказівник типу **void** і його оголошують так:

```
void* <ідентифікатор> ;
```

Оскільки **void** не має типу, то до нього не може бути застосована операція розіменування (взяття вмісту) і адресна арифметика, так як невідомо уявлення даних.

6.2 Операції із вказівниками

Після того, як було розглянуто концепцію вказівника як змінної, що зберігає адресу, час навчитися керувати цим інструментом. Операції із вказівниками – це набір фундаментальних дій, які дозволяють маніпулювати адресами та отримувати доступ до даних, на які ці адреси вказують.

Уся робота з покажчиками зводиться до двох ключових, протилежних дій, для виконання яких мова С надає два **фундаментальні унарні оператори**:

1. **Отримання адреси** змінної - **&** - застосовується до об'єктів, які мають власне ім'я та розміщені в оперативній пам'яті.
2. **Розіменування вказівника** (*****) – для доступу до змінної, адреса якої зберігається у вказівнику. Цю операцію можна використовувати як для отримання, так і для зміни значення змінної, яка не є константою.

Символ «*» в оголошенні вказівника та в **операції розадресації** – **різні речі**, але позначаються однаковим символом

Існує кілька способів **ініціалізації** вказівника. Перший – присвоєння вказівнику адреси існуючого об'єкта (за допомогою операції отримання адреси або за допомогою значення іншого ініціалізованого вказівника).

Вказівнику можна присвоїти значення іншого вказівника, якщо вони є одного типу, інакше використовують операцію приведення типу. Вказівник може бути константою, змінною або вказувати на константу чи змінну.

Щоб отримати значення, записане в деякій області, на яке посилається вказівник потрібно скористатися операцією **розіменування** вказівника *****. Для

цього необхідно поставити зірочку перед ім'ям і отримаємо доступ до значення покажчика.

Приклад. Розіменування вказівника. Наведено такий фрагмент програми:

```
int X = 134;
int* ptr = &X;// Ініціалізуємо вказівник адресою змінної X
int  Y  =  *ptr;//  Набуття  значення  величини,  адреса  якої
зберігається в ptr
```

В цій програмі оператор **`int Y = * ptr`** здійснює: операцію розіменування покажчика; операцію присвоювання. Операція розіменування має вищий пріоритет у порівнянні з операцією присвоєння, і тому виконується першою. Дію оператора **`int Y = * ptr`** можна умовно уявити як послідовне виконання наступних дій:

1. читати адресу, збережену в покажчику **`ptr`** (**`00AF`**);
2. отримати значення, що знаходиться за цією адресою (**`134`**);
3. присвоїти змінній **`Y`** результат попередньої дії.

Розглянемо програму, яка використовуватиме покажчики (див. рис. 6.3).

```
#include <stdio.h>

int main()
{
    int var = 123; // ініціалізація змінної var числом 123
    //присвоїли адресу змінної var вказівнику ptrvar
    int* ptrvar = &var;
    //адресу змінної var отримано операцією взяття адреси
    printf(" &var    = %p\n", &var);
    //адреса змінної var є значенням вказівника ptrvar
    printf(" ptrvar  = %p\n", ptrvar);
    //значення змінної var
    printf(" var     = %d\n", var);
    //значення var отримано через розіменування вказівника ptrvar
    printf(" *ptrvar = %d\n", *ptrvar);
    return 0;}
```

Рисунок 6.3 – Лістинг програми з покажчиками

На рисунку 6.4 наведено результат роботи програми наведної на рисунку 6.3

```
&var    = 004FF810
ptrvar  = 004FF810
var     = 123
*ptrvar = 123
```

Рисунок 6.4 – Результат роботи програми

Як видно з результату роботи програми, доступ до значення змінної **`var`** отримано через безпосереднє звернення до неї та через розіменування вказівника **`ptrvar.`**

Розіменування неініціалізованого вказівника або вказівника зі значенням 0 (NULL) є невиправною помилкою часу виконання, оскільки при цьому використовується **дикий вказівник** – змінна типу вказівник, якій не присвоєне значення адреси жодного об’єкта, але з якою програма намагається працювати як із вказівником на дані. Фрагмент програми демонструє ситуацію виникнення дикого вказівника:

```
int y, *p;
y = *p; //ПОМИЛКА, *p вказує на невідому ділянку пам'яті
*p = 10; //число 10 записується у невідому ділянку пам'яті
```

Якщо вказівник ще не вказує ні на який об'єкт, його слід ініціалізувати константою **`NULL`**. Ця константа чітко вказує, що вказівник **нікуди не вказує**.

Використання **`NULL`** є ключовим запобіжником, оскільки це дозволяє програмісту перевірити вказівник перед розіменуванням:

```
int* ptr;            // НЕПРАВИЛЬНО! Дикий вказівник.
int* ptr_safe = NULL; // ПРАВИЛЬНО! Вказівник безпечний.

// Безпечна перевірка перед розіменуванням
if (ptr_safe != NULL) {
    *ptr_safe = 10;
     }
```

Рисунок 6.5 – Перевірка вказівника

Мова С не містить вбудованих механізмів захисту (запобіжників) від некоректної роботи з пам’яттю. Транслятор С не перевіряє, чи є покажчик дійсним перед його використанням. Отже, вся відповідальність за цілісність пам’яті покладається на програміста.

6.3 Арифметика вказівників

Арифметика вказівників – це набір операцій, які дозволяють маніпулювати адресою, що зберігається у вказівнику, пересуваючи його по пам’яті. Це не просто додавання чисел; це пересування між елементами певного типу даних.

З **арифметичних операцій**, найчастіше використовуються операції додавання, віднімання, інкремент і декремент, оскільки за допомогою цих операцій, наприклад в масивах, обчислюється адреса наступного елементу.

При додаванні або відніманні цілого числа **`N`** від вказівника, адреса, що зберігається в ньому, змінюється **кратно розміру базового типу** даних у байтах. Це гарантує, що при додаванні 1 вказівник завжди переходить до **наступного логічного елемента**, незалежно від того, чи займає він 1 байт (**`char`**) чи 8 байтів (**`double`**).

Загальна формула для обчислення нового значення адреси при додаванні (відніманні) цілого числа N:

```
<нова_адреса>=<стара_адреса>+/-<розмір_базового_типу> *N
```

Рисунок 6.6 – Формула обчислення значення вказівника при додаванні (відніманні) до нього цілого числа **N**

В цій формулі розмір базового типу визначається функцією **sizeof(тип)**. Цей механізм дозволяє вказівникам легко переходити від одного елемента в масиві до наступного, незалежно від того, скільки байтів займає елемент.

При додаванні цілого числа **n** до вказівника **ptr** переміщує вказівник на **n** елементів вперед у пам'яті. На рисунку наведено приклад обробки масиву через додавання до вказівника цілого числа:

```
int arr[] = { 10, 20, 30 };
int* p = arr; // p вказує на arr[0] (адреса 1000, наприклад)

// p + 2 переміщує вказівник на 2 * sizeof(int) байти вперед (1000
+ 8 = 1008)
int* p_next = p + 2;
// *p_next тепер дорівнює 30
// Якщо p_next вказує на arr[2] (адреса 1008)
int* p_prev = p_next - 2;
// *p_prev тепер дорівнює 10
```

Рисунок 6.7 – Додавання та віднімання цілого числа від вказівника

При відніманні цілого числа **n** від вказівника відбувається переміщення на **n** елементів назад у пам'яті.

Віднімання двох вказівників, що вказують на елементи **одного й того ж масиву**, повертає ціле число, яке показує **кількість елементів** між ними (див. рис. 6.8).

```
int array[] = { 10, 20, 30, 40, 50 };
int* start = array;         // Вказує на 10
int* end = &array[4];       // Вказує на 50

// Результат: (Адреса кінця - Адреса початку) / sizeof(int) = 4
long distance = end - start;
     printf("Відстань у елементах: %ld\n", distance); //Виведе 4
```

Рисунок 6.8 – Віднімання вказівників

Ім'я масиву являє собою константний вказівник на початок (на нульовий елемент) масиву, тобто наступні записи є еквівалентними:

```
array == & array == & array [0]
```

Рисунок 6.9 – Вказівник на масив

6.4 Вказівники на вказівники

Вказівники можуть посилатися на інші вказівники. При цьому в комірках пам'яті, на які посилатимуться перші вказівники, міститимуться не значення, а адреси інших вказівників. Число символів * при оголошенні вказівника показує порядок вказівника. Щоб отримати доступ до значення, на яке посилається вказівник його необхідно розіменовувати відповідну кількість разів. Розглянемо програму, яка виконуватиме деякі операції з вказівниками порядком вище першого (див. рис. 6.10):

```
int main()
{
    int var = 123; // ініціалізація змінної var числом 123
    int* ptrvar = &var; //покажчик на змінну var
    int** ptr_ptrvar = &ptrvar; // покажчик на покажчик на var
    int*** ptr_ptr_ptrvar = &ptr_ptrvar;
    printf("  var           = %d\n", var);
    printf("  *ptrvar       = %d\n", *ptrvar);
// двічі розіменовуємо покажчик, оскільки він другого порядку
    printf("  **ptr_ptrvar  = %d\n", **ptr_ptrvar);
// покажчик третього порядку
    printf("  ***ptr_ptrvar = %d\n", ***ptr_ptr_ptrvar);
    printf("\n***ptr_ptr_ptrvar -> \t**ptr_ptrvar ->    \t*ptrvar
->  \t\tvar -> %d\n", var);
    printf("%p -> %p -> %p -> %p -> %d", &ptr_ptr_ptrvar,
&ptr_ptrvar, &ptrvar, &var, var);
    printf("\n");
    return 0;}
```

Рисунок 6.10 – Вказівники різних порядків

6.5 Вказівники на функції

Вказівники можуть посилатися на функції. Ім'я функції, як і ім'я масиву само по собі є вказівником, тобто містить адресу входу (див. рис. 6.11).

```
<тип> (*<ідентифікатор покажчика>)(<аргументи функції>)
```

Рисунок 6.11 – Оголошення вказівника

Символ вказівника і його ім'я беруться в круглі дужки, щоб показати, що це вказівник, а не функція, яка повертає вказівник на певний тип даних. Після імені вказівника розташовують круглі дужки, де перераховуються всі аргументи через кому як в оголошенні прототипу функції. Аргументи успадковуються від тієї функції, на яку посилатиметься вказівник.

Розглянемо програму (див. рис. 6.12), яка використовує вказівник на функцію. Програма знаходить НСД – найбільший спільний дільник. Вхідні числа також повинні бути цілими числами.

```
int gcd(int, int); // прототип вказаної функції
```

```
int main(){
     int (*ptrgcd)(int, int); //оголошення вказівника на функцію
     ptrgcd = gcd; // ініціалізуємо вказівник ptrgcd адресою
функції gcd
     int a, b;
     printf ("Enter first number:  ");
     scanf_s("%d", &a);
     printf("\nEnter second number:  ");
     scanf_s("%d", &b);
     printf("\nGCD = %d \n", ptrgcd(a, b)); // виклик функції через
вказівник
     return 0;}
int gcd(int number1, int number2) // рекурсивна функція обчислення
НСД
{
     if (number2 == 0) // умова виходу з рекурсії
          return number1;
     return gcd(number2, number1 % number2); }
```

Рисунок 6.12 – Лістинг програми обчислення НСД

6.6 Динамічні масиви в C/C++

Якщо масив створюється під час оголошення – пам'ять для нього розподіляється під час компіляції і не змінюється до закінчення роботи програми. Такий розподіл пам'яті для масиву називається **статичне зв'язування**. За допомогою певних функцій можливо створити масив під час виконання програми, якщо в цьому є необхідність (наприклад, користувач вказує розмірність масива під час роботи програми). Такий розподіл пам'яті для масиву називається **динамічне зв'язування**, а самі масиви – **динамічними масивами**.

При статичному зв'язуванні необхідно задати розмір масиву під час написання програми. При динамічному зв'язуванні програма може прийняти рішення про розмір масиву під час роботи (наприклад, на основі введеної користувачем інформації).

В C/C++ операції **new** і **delete** призначені для динамічного розподілу пам'яті комп'ютера. Операція **new** виділяє пам'ять з області вільної пам'яті, а операція **delete** звільняє виділену пам'ять. Пам'ять, яка виділяється, після її використання повинна вивільнятися, тому операції **new** і **delete** використовуються парами (див. рис. 6.13). Навіть якщо не вивільняти пам'ять явно, то вона звільниться ресурсами ОС по завершенню роботи програми. Рекомендується все ж таки не забувати про операцію **delete**.

```
<тип> *<ідентифікатор вказівника> = new <int>;
delete ptrvalue;
```

Рисунок 6.13 – Створення динамічної змінної

Найчастіше операції **new** і **delete** застосовуються для створення динамічних масивів, а не для створення динамічних змінних. Розглянемо фрагмент коду створення одновимірного динамічного масиву (див. рис. 6.14).

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

int main()
{
     srand(time(0));
     float* ptrarray = new float[10]; //створення масиву з 10
дійсних чисел
     for (int count = 0; count < 10; count++) {
       // заповнення масиву випадковими числами від 1 до 10
       //звернення до елементів масиву через вказівник
       ptrarray[count] = (rand() % 10 + 1) / float((rand() % 10 +
1));
     }
     for (int count = 0; count < 10; count++) {
       printf(" ptrarray[%d] =  %.2f, %p \n", count,
ptrarray[count]);
     }
     delete[] ptrarray; // вивільнення пам'яті
     return 0;}
```

Рисунок 6.14 – Лістинг програми генерування динамічного масиву

Для друкування адрес елементів масиву на екран необхідно застосувати специфікатор формату **%p**, а в якості аргумента до функції **printf()** передати **&ptrarray[count]**.

При передачі масивів до функцій, треба пам'ятати:

1. До функції масиви завжди передаються через адресу початку масиву;
2. Ім'я масиву являє собою константний покажчик на початок (на нульовий елемент) масиву, тобто наступні записи є еквівалентними:

mas == &mas == &mas[0]

Тепер розглянемо фрагмент коду, в якому показано, як оголошується двовимірний динамічний масив. Оголошення та видалення двовимірного динамічного масиву виконується за допомогою циклу, тому розглянемо програму, в якій створюється двовимірний динамічний масив (див. рис. 6.15).

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

int main()
{
```

```
     srand(time(0));
     int row = 4, column = 5;
     float** ptrarray = new float* [row]; // створення масиву
вказівників на рядки
     for (int i = 0; i < row; i++)
          ptrarray[i] = new float[column]; // створення масиву
стовпців
     for (int i = 0; i < row; i++)
          for (int j = 0; j < column; j++)
          ptrarray[i][j] = (rand() % 10 + 1) / float((rand() %10 + 1));
     printf(«\n»);
     for (int i = 0; i < row; i++)
     {
          for (int j = 0; j < column; j++)
               printf(« %.2f «, ptrarray[i][j]);
          printf(«\n»);
     }
     // вивільнення пам'яті
     for (int i = 0; i < row; i++)
          delete[] ptrarray[i];
     return 0;}
```

Рисунок 6.15 – Генерування двовимірного динамічного масиву

Спочатку оголошується покажчик другого порядку **`float **ptrarray`**, який посилається на масив покажчиків **`float* [row]`**, де розмір масиву дорівнює чотирьом. Після чого в циклі **`for`** кожному рядку масиву оголошеного в рядку 2 виділяється пам'ять під п'ять елементів. В результаті отримуємо двовимірний динамічний масив **`ptrarray[4][5]`**.

Тести для самоконтролю

1. Який оператор використовується для отримання адреси змінної для присвоєння її вказівнику?

А. &

Б. ?

В. *

2. Що поверне операція p + 3, якщо p є вказівником на int (4 байти) і містить адресу 1000_{16}?

А. Адресу 1012_{16}.

Б. Адресу 1003_{16}.

В. Значення елемента на позиції p[3].

3. Яка функція в мові С використовується для виділення блоку пам'яті для динамічного масиву?

А. delete.

Б. sizeof().

В. malloc().

4. Яке з наступних тверджень правильно описує вказівник на вказівник?

А. Він зберігає адресу іншого вказівника (int *p).

Б. Він зберігає значення змінної типу int.

В. Він може вказувати лише на останній елемент масиву.

5. Яке правило застосовується при відніманні двох вказівників?

А. Результатом є різниця їхніх адрес у байтах.

Б. Результатом є кількість елементів між двома адресами.

В. Операція не дозволена.

6. Яка функція в мові С використовується для звільнення раніше виділеної динамічної пам’яті?

А. free().

Б. realloc().

В. calloc().

7. Що таке розіменування вказівника?

А. Присвоєння вказівнику нової адреси.

Б. Використання оператора & для отримання адреси змінної.

В. Використання оператора * для доступу до значення, що зберігається за адресою, на яку вказує вказівник.

8. Яка арифметична операція не дозволена для вказівників?

А. Додавання двох вказівників (ptr1 + ptr2).

Б. Інкремент (ptr++).

В. Віднімання цілого числа (ptr - N).

9. Для чого в програмуванні на С використовується вказівник на функцію?

А. Для динамічної зміни аргументів функції під час виконання.

Б. Для виклику функції, адреса якої визначається під час виконання.

В. Для збереження повертаючого значення функції.

10. Яка проблема виникає, якщо програміст не звільняє динамічно виділену пам’ять за допомогою free()?

А. Помилка компіляції.

Б. Витік пам’яті.

В. Помилка сегментації.

Практичні завдання

1. Напишіть програму, яка динамічно виділяє пам’ять для масиву, що містить 5 цілих чисел (int), використовуючи функцію malloc(). Не забудьте про sizeof(). Використовуйте арифметику вказівників замість індексів.

2. Заповніть динамічний двовимірний масив псевдовипадковими числами в діапазоні від -10 до 10. Використовуючи вкладені цикли for, виведіть матрицю на екран у вигляді таблиці, щоб кожен рядок відображався на новому рядку консолі. При виведення матриці використовувати вказівник, а не індексну форму.

Контрольні запитання

1. У чому полягає відмінність між оператором взяття адреси та оператором розіменування?

2. Чим арифметика вказівників відрізняється від звичайної цілочисельної арифметики?

3. Що таке витік пам’яті та в чому його небезпека?

4. Скільки разів необхідно виконати розіменування вказівника третього порядку, щоб отримати доступ до кінцевого значення змінної?

5. Як у мові С пов’язані масиви та вказівники?

Тема 7. Символи та рядки

План

1. Представлення символів в С.
2. Представлення рядків в С.
3. Функції для обробки рядків.

7.1 Представлення символів в С

Множина допустимих значень символьного (літерного типу) – це множина цілих чисел в діапазоні від 0 до 255. Зберігання значення такого типу потребує одного байта оперативної па'яті. Найчастіше такий тип застосовується для зберігання символів кодової таблиці **ASCII**, а отже, даними цього типу є окремі символи. Кожному символу відповідає ціле число (код) в діапазоні від 0 до 127. Як вже говорилось раніше, символьний тип позначається ідентифікатором **char**.

Коли символьні змінні отримують певні значення, то комп'ютер зберігає в пам'яті не символи, а їхні коди. Тому, якщо символьній змінній присвоїти певне ціле число, то компілятор сприйме його як код певного символу з **ASCII**-таблиці. Це правила поширюється лише на цілі числа. Зважаючи на різні кодування розширеної частини **ASCII**-таблиці, для уникнення помилок при роботі з символами, потрібно використовувати їхнє символьне подання, а не коди (див. рис. 7.1).

Значення символьного типу записуються в одинарних лапках. Наприклад, **`A`**, **`1`**, **`+`** тощо. На рисунку 7.1 наведено фрагмент програми, який демонструє зв'язок між типами **int** та **char**.

```
    char symbol = 'd';
    int a = symbol;//це символ 'd' масиву symbol
    printf("%d\n", a); //буде надруковано 100, оскільки в ASCII-
таблиці код символа 'd'дорівнює 100
    printf("%c\n", symbol); //буде надруковано d
```

Рисунок 7.1 – Зв'язок між типами даних **int** та **char**

Далі, в таблиці 7.1 наведено фрагмент **ASCII**-таблиці з відповідностями між певними символами та їх кодами

Таблиця 7.1 – Відповідність між символом та його кодом

Символ	Код
` ` (пробіл)	32
`0`… `9`	48 … 57
`A` … `Z`	65 … 90
`a` … `z`	97 … 122

Перелік деяких керуючих послідовностей, котрі називають ESC-послоідовностями, подано в таблиці 7.2.

Таблиця 7.2 – Деякі керуючі символи таблиці **ASCII** кодів

Символ	Опис
\n	Символ переведення курсору на початок наступного рядку
\r	Перехід на початок поточного рядка (переведення каретки)
\t	Горизонтальна табуляція (відповідає клавіші <Tab>)
\v	Вертикальна табуляція
\b	Символ вилучення попереднього символу перед курсором (відповідає клавіші <BackSpace>)
\a	Символ звукового сигналу системного динаміка
\\	Зворотна коса риска
\?	Знак питання
\`	Апостроф (одинична лапка)
\”	Подвійні лапки
\0	Завершальний символ рядка символів (нуль-символ)
\0oo	Вісімковий код символу, де oo – одна, дві чи три вісімкові цифри (0…7)
\0xhh	Шістнадцятьковий код символу, де hh – одна, дві чи більше шістнадцятькових цифр (0…9, A, …, F)

Для даних символьного типу означено операції порівняння. Символи вважають рівними, якщо рівні їх **ASCII**-коди. Один символ вважають більшим за інший, якщо його **ASCII**-код більший. Крім операцій порівняння для даних символьного типу означено операції додавання та віднімання, у результаті виконання яких утворюються нові символи. Ці операції позначають символами «+» та «-».

Приклад. Отримати символ десяткової цифри зі значення цілої змінної, що належить діапазону 0…9.

Отримати такий символ можна таким чином:

```
int n; char c; c = n + '0';
```

Код символа `0` дорівнює 48, а тому додаючи до числа **n** (приймає значення від 0 до 9) число 48 отримуємо число від 48 до 57. А оскільки тип змінної **c** (результат такого додавання) – **char**, то отримане число це **ASCII**-код символів `0`…`9`. А обернене перетворення можна зробити за допомогою такої комбінації: `if (c >= '0' && c <= '9') n = c - 0;`

Приклад. Перетворити маленьку латинську літеру на велику.

Зробити це можна таким чином:

```
if (c >= 'a' && c <= 'z') c = c - 'a' + 'A';
```

В даному фрагменті програми перш за все перевіряється умова, чи є значення змінної маленькою латинською літерою. Потім отримуємо десіткове число (від 0 до 9), а далі додаємо код символа `A` і таким чином отримуємо велику літеру відповідну до маленької літери.

7.2 Представлення рядків в С

Рядковий тип даних в мові програмування С реалізовано як лінійний масив символів. Для полегшення роботи з рядковими даними в С реалізовано ряд функцій. Кожний символ рядка зберігається в масиві типу **char**. При цьому, останній символ повинен бути **'\0'**.

Далі на рисунку 7.2 наведено декілька прикладів оголошення рядків

```
#include <stdio.h>

int main()
{

    char name1[15];
    char name2[15] = "Coding C";
    char name3[] = { 'a', 'b', 'c', 'd', 'e' };
    char name4[15]={'C', 'o', 'd', 'i', 'n', 'g', ' ', 'C'};
    char S[256] = "";
    printf("%s\n", name1);
    printf("%s\n", name2);
    printf("%s\n", name3);
    printf("%s\n", name4);
```

Рисунок 7.2 – Приклад оголошення рядків в програмі

Наведемо в таблиці 7.3 опис декількох функцій для обробки символів:

Таблиця 7.3 – Функції для обробки символів

Прототип функції	Опис
`int isalnum(int c);`	Перевірка, чи є символ літерою або цифрою
`int isalpha(int c);`	Перевірка, чи є символ літерою
`int iscntrl(int c);`	Перевірка, чи є символ керуючим
`int isdigit(int c);`	Перевірка, чи є символ десятковою цифрою
`int isgraph(int c);`	Перевірка, чи є символ видимим
`int isprint(int c);`	Перевірка, чи є символ видимим, включаючи пробіл
`int islower(int c);`	Перевірка, чи є символ літерою нижнього регістру
`int ispunct(int c);`	Перевірка, чи є символ знаком пунктуації
`int isspace(int c);`	Перевірка, чи є символ пробільним
`int isupper(int c);`	Перевірка, чи є символ літерою верхнього регістру
`int isxdigit(int c);`	Перевірка, чи є символ шістнадцятьковою цифрою
`int tolower(int c);`	Перетворення символу в нижній регістр
`int toupper(int c);`	Перетворення символу у верхній регістр

Вищезазначені функції перевірки й перетворення регістру працюють лише з латинськими літерами. Спроба використати ці функції для символів кирилиці спричинить виведення повідомлення про помилку. Тому для роботи з літерами кирилиці слід використовувати перевірку належності символу до

відповідного символьного проміжку, а для перетворення регістру – зменшення чи то збільшення коду символу на різницю кодів між великими та малими літерами (для більшості літер вона становить 32).

7.3 Функції обробки рядків

Для введення одного символу в консолі існує функція **getch** з бібліотеки **conio.h**. Для введення і виведення рядків в консолі використовують такі функції **gets/puts** та **scanf/printf** (з бібліотеки **stdio.h**).

Таблиця 7.4 – Функції обробки рядків

Функція	Опис функції
`puts(s)`	виводить рядок **s** на екран, замінюючи нуль-символ на Enter.
`gets_s(s, buf)`	зчитує символи з клавіатури до появи символу переведення рядка Enter і записує їх у рядок **s**, параметр **buf** відповідає за максимальну кількість символів, які буде зчитано
`printf("%s", S)`	виводить рядок **S** на екран
`scanf_s("%s", S, buf)`	зчитує символи з клавіатури і записує їх у рядок **S**, параметр **buf** відповідає за максимальну кількість символів, які буде зчитано
`strlen(char *s);`	повертає довжину рядка (без урахування символу закінчення рядка)
`strcpy(char *s1, char *s2);`	копіює в **s1** рядок **s2**, повертає **s1**, при цьому старе значення **s1** втрачається
`strncat(char *s1, char *s2, size_t n);`	приєднує до рядка **s1** перші **n** символів з рядка **s2**
`strpbrk(char *s1, const char *s2)`	знаходить місце першого входження в рядок **s1** будь-якого символу рядка **s2** і повертає частину рядка **s1**, починаючи з цього символу і до кінця
`strtok(char *s1, const char *s2);`	повертає частину (лексему) рядка **s1** від поточної позиції вказівника до розділового символу, зазначеного у рядку **s2**

Наведемо декілька прикладів роботи з рядками.

Приклад. Обчислити довжину рядка.

Текст програми наведено на рисунку 7.3

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

int main()
{
    char S[6] = "Hello"; // 7 байт заповнено літерами, 8-й байт - нуль-
символ '\0'
    int lenght = strlen(S);
    printf("Number of symbols in string S %d", lenght);
    return 0;
}
```

Рисунок 7.3 – Обчислення кількості символів у рядку

Приклад. Вивести всі символи та коди таблиці ASCII

На рисунку 7.4 наведено одну з можливих реалізацій:

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

int main()
{
    for (int i = 32; i < 256; i++) {
        if (i % 4 == 0) printf("\n");
        printf("%3i = %c \t", i, (char)i);
    }
    return 0;
     }
```

Рисунок 7.4 – Виведення таблиці ASCII на екран

Приклад. Розташувати символи рядка в порядку зростання їхніх кодів. Обчислити кількість пробілів.

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

int main()
{
    char testString[256] = "";//оголошення рядка
    int cnt = 0; //змінна для кількості пробілів
    printf("Enter your string ...");
    gets_s(testString);//зчитування рядка
    printf("You input this string ...%s\n", testString);
    int lenght = strlen(testString); //обчислення довжини введеного рядка
    for (int i = 0; i < lenght; i++) {
        if (testString[i] == ' ')//або так testString[i]==32
            cnt++;
     }
    printf("number of spaces is %d", cnt);
    //сортування символів рядка за зростанням їх кодів
```

```
    for (int i = 0; i < lenght; i++) {
         for (int j = 0; j < lenght - i - 1; j++) {
                 if (testString[j] > testString[j + 1]) {
                     char temp = testString[j];
                     testString[j] = testString[j + 1];
                     testString[j + 1] = temp;
                 }
         }
     }
     printf("\nString after sorting ... %s\n\n", testString);
}
```

Рисунок 7.5 – Сортування елементів рядка та обчислення кількості пробілів

Приклад. Користувач задає інвертований рядок (тобто записаний справа наліво), а комп‘ютер відображає правильний рядок та рахує кількість слів у ньому. Словом вважається набір літер між двома пробілами.
Реалізація даної задачі має вигляд (див. рис. 7.6)

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

int main()
{
    char S[256], newS[256];//змінні, що містять введений рядок та
результат
    printf("Input Reverse String...\n");
    gets_s(S);//зчитали рядок з клавіатури
    printf("You input this string: %s \n", S);
    char* next_token = NULL;

    //блок інвертування рядка
    int n = strlen(S);//визначення довжини рядка
    for (int i = 0; i < n; i++)
        newS[i] = S[n - i - 1];//цикл, що інвертує заданий рядок
    newS[n] = '\0'; //обов'язковий символ закінчення рядка
    printf("The Result String is -%s- \n", newS);
    char *word;//покажчик на початок слова
    word = strtok_s(newS, " ", &next_token);//виділяємо слово між
пробілами
    while (word != NULL) {
        printf("\n\t%s", word);
        word = strtok_s(NULL, " ", &next_token);
    }
    return 0;
}
```

Рисунок 7.6 – Лістинг програми інвертування рядка

Приклад. Видалити у рядку всі слова, які закінчуються на голосну.
Одна з можливих реалізацій даного завдання наведена на рисунку 7.7.

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

```
int main()
{
    while (1)
    {
        char S[256], newS[256] = {0}; // оголошуємо два рядки
        char vowels[] = "aeiouy";//створюємо масив з голосними
        int i, lennewS; //змінна для довжини нового рядка
        char *next_token = NULL;

        printf("  \nInput  string: \n");
        gets_s(S);
        printf("  You input this string: \n%s\n", S);
        char* word; //покажчик на адресу пам'яті, де зберігатиметься
слово word
        word = strtok_s( S , " ", &next_token); // виділяємо слово, тобто
послідовність символів між пробілами
        while (word != NULL)// поки не досягнуто кінець рядка
        {
            char isvowel = 0;  //змінна, яка сигналізуватиме, чи
закінчується слово на голосну
            int lenw = strlen(word);  // довжину вирізаного з рядка слова
зберігаємо в змінній lenw (знадобиться пізніше)
            for (i = 0; i < sizeof(vowels); i++){
                if (word[lenw - 1] == vowels[i]) //послідовно порівнюємо
останню букву слова з елементами масива vowels
                {
                    isvowel = 1; // якщо це так, то слово нам не цікаве,
робимо присвоєння і break
                    break; // переходимо на рядок 39, а далі на 24
                }
            }
            if (!isvowel) // якщо зустріли слово з приголосною
наприкінці, то формуємо з них рядок без голосного закінчення
            {printf("Add to rezult string this word: %s\n ", word);
             strcat_s(newS, word);    //об'єднуємо рядок newS і рядок
word
             strcat_s(newS, " ");     //додаємо пробіл, щоб слова не
поєднувались при друці
            }
            word = strtok_s(NULL, " ", &next_token); // переходимо до
наступного слова
        }
        lennewS = strlen(newS); //знаходимо довжину нового рядка, щоб на
правильну позицію поставити символ кінця рядка
        newS[lennewS] = '\0';
        printf("\n  New string without vowels in the end of words  \n");
        printf("\n %s \n", newS);
        _getch();
    }
    return 0;}
```

Рисунок 7.7 – Лістинг програми видалення з рядка слів

Декілька коментарів до цієї програми. По-перше, задачу про видалення слів з рядку **S** переформульовано у задачу формування нового рядка **newS** , який складатиметься зі слів без голосної в кінці слова. Фрагмент програми,

який відповідає за порівняння останньої букви кожного слова з елементами масиву **vowels** можна замінити відповідною бібліотечною функцією **strrchr()** з бібліотеки **string.h**.

Зауваження! Для коректної роботи програми необхідно підключити наступні модулі: **<stdio.h>, <stdlib.h> та <string.h>.**

Тести для самоконтролю

1. Що таке рядок у мові С?
   А. Вказівник на перший елемент структури.
   Б. Спеціальний тип даних string.
   В. Масив символів, завершений нульовим символом \0.
2. Яка функція з бібліотеки <string.h> використовується для копіювання одного рядка в інший?
   А. strcat().
   Б. strcpy().
   В. strcmp().
3. Який символ використовується для завершення рядка у С?
   А. Пробіл.
   Б. Нульовий символ.
   В. Символ нового рядка.
4. Як оголошується змінна, що зберігає один символ ('a')?
   А. int i = 'a';
   Б. string s = 'a';
   В. char c = 'a'.
5. Яка функція з <string.h> використовується для визначення довжини рядка?
   А. strlen().
   Б. size_of().
   В. strcmp()
6. Який тип даних найчастіше використовується для представлення рядка у функціях С (наприклад, як параметр функції printf()?
   А. Вказівник на int.
   Б. Вказівник на char.
   В. Звичайна змінна char.
7. Який результат повертає функція strcmp(str1, str2), якщо рядок str1 лексикографічно більший за рядок str2?
   А. Додатне число.
   Б. 0.
   В. Від'ємне число.
8. Який тип лапок (одинарні чи подвійні) використовується для визначення рядкової коснтанти?
   А. Одинарні лапки (' ').
   Б. Символ $.
   В. Подвійні лапки (" ").

9. Яка функція використовується для об’єднання двох рядків?

А. strcat().

Б. strlwr().

В. strchr().

10. Який ефект матиме вираз char c = 65?

А. Змінна c зберігатиме код ASCII 65, що відповідає символу 'A'.

Б. Змінна c зберігатиме число 65, але інтерпретуватиметься як символ '1'.

В. Виникне помилка, оскільки char не може зберігати числа.

Практичні завдання

1. Напишіть програму, яка об’єднує два рядки. Виведіть на екран об’єднаний рядок.

2. Напишіть програму, яка використовує функції для аналізу та копіювання рядків. Визначте довжину утвореного рядка за допомогою функції strlen() та виведіть її на екран.

Контрольні запитання

1. Який ключовий символ обов’язково повинен завершувати рядок?

2. У чому полягає принципова відмінність між результатом функцій strlen(myString) та sizeof(myArray)?

3. В чому полягає призначення функції strcmp()?

4. Яку обов’язкову перевірку має виконати програміст перед викликом функцій strcpy() та strcat()?

5. Який тип даних використовується для зберігання символів?

Тема 8. Функції користувача

План

1. Поняття про функцію та її складові
2. Функції типу void
3. Функції, що повертають значення
4. Передавання параметрів у функцію
5. Рекурсивні функції

8.1 Поняття про функцію та її складові

Програмний код у великих програмах часто містить частини, що повторюються. Використання функцій дає можливість уникнути створення двох чи більшої кількості схожих фрагментів програмного коду для розв'язання однакових завдань із різними вхідними даними. Як наслідок, програма стає більш зрозумілою і простішою у відлагодженні.

Функція – поіменована послідовність описів та операторів, яка виконує деяку скінченні послідовність дій. Будь-яка функція складається із: заголовка (оголошення функції) і тіла (визначення функції).

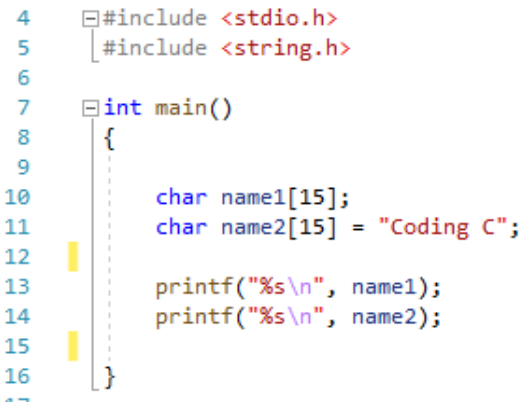
Кожна С-програма містить принаймні одну функцію – функцію **`main()`**. Крім неї, програма може містити необмежену кількість функцій, створених користувачем.

Існує два основні види функцій: **стандартні** та створені користувачем (**користувацькі**). Будь-яка функція повинна бути оголошена і визначена. Оголошення функції повідомляє компілятору як її викликати, а визначення – описує її тіло.

Виконання функції починається тоді, коли в тексті програми зустрічається оператор виклику цієї функції.

Виклик функції завжди позначається ім'ям функції та круглими дужками, в яких стоять змінні (константи, вирази), значення яких передаються (підставляються) замість аргументів. У випадку, коли функція не має аргументів, круглі дужки все одно є обов'язковими.

На рисунку продемонстровано виклик стандартної функції **`printf()`**. Бачимо круглі дужки, список параметрів, які передаємо. Прототип цієї функції міститься в заголовному файлі **`stdio.h`**, а визначення цієї функції зберігається в стандартних бібліотеках С, які постачаються разом з компілятором. При обробці програми компілятор бачить прототип функції у відповідному заголовному файлі (вірніше навіть відбувається копіювання всього вмісту цього заголовного файлу), а при лінкуванні програма буде підключена до стандартної бібліотеки, де знаходиться готова реалізація цієї функції (тобто правило роботи з нею).



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

int main()
{

    char name1[15];
    char name2[15] = "Coding C";

    printf("%s\n", name1);
    printf("%s\n", name2);

}
```

Рисунок 8.1 – Виклик стандартної функції **`printf()`**

Навіщо ж створювати власні функції? Серед основних чинників варто виділити:

1. **Повторне використання коду.** Замість того, щоб копіювати один і той самий блок інструкцій у різних місцях, можна написати його один раз у функції й викликати там, де потрібно.

2. **Зручність і структурованість.** Програма структурується на логічні частини. Наприклад, окрема функція для обчислення факторіала, окрема – для перевірки парності числа. Це робить код більш зрозумілим.

3. **Масштабованість.** Якщо потрібно змінити алгоритм, можна редагувати лише функцію, а не всі модулі (компоненти програми), де цей код використовується.

4. **Тестування та відлагодження.** Легше перевіряти неве7ликі фрагменти коду (функції), ніж одразу всю програму. Це підвищує надійність.

5. **Модульність і командна робота.** У великих проєктах фахівці можуть працювати над різними функціями незалежно, а потім інтегрувати їх у спільну систему.

6. **Читабельність.** Функція з назвою **`calculateAverage()`** одразу пояснює її призначення, навіть без перегляду її внутрішнього коду.

Оголошення функції описується за допомогою прототипу і має бути описане перед її викликом. **Прототип** функції не містить інформації про її внутрішню будову, він описує, як до неї звертатись. Для **виклик**у функції транслятору достатньо знати лише її ім’я, тип результату, кількість і типи аргументів. Прототип функції має такий формат:

```
<типПовернення> Ім'яФункції(СписокПараметрів);
```

Рисунок 8.2 – Прототип функції

Тут *ТипПовернення* – тип даних, що повертається функцією. Якщо він не зазначений.

СписокОголошенихПараметрів задає тип та ім’я кожонго з параметрів функції, розділених комами (ім’я параметра можна опускати). Список параметрів функції можу бути порожнім.

Визначення функції складається з її заголовка і послідовності операторів (тіло функції), вкладених у фігурні дужки. Якщо функція повретає будь-яке значення, в тілі функції обов'язково повинен бути присутнім оператор повернення (**return**) з параметром того ж типу.

```
<типПовернення> Ім'яФункції(СписокОголошенихПараметрів)
{
    //тіло функції
}
```

Рисунок 8.3 – Визначення функції

Виклик функції – зазначення ідентифікатора функції (її імені), за яким в круглих дужках наведено список **аргументів**, розділених комами.

Параметри (*СписокОголошенихПараметрів*), що вказуються в заголовку функції при її описі – **формальні параметри.**

Параметри, що вказуються при виклику функції, називаються **фактичними параметрами (аргументами)**. На рисунку 8.4 фактичними параметрами, наприклад, є числа 52,5, 123,5, до яких застосовується функція **max**.

На рисунку 8.4 наведено приклад посалідовного виклику декількох функцій.

```
double maxValue = max(52.5, 123.5);
int j = swap(10, 2);
func();
```

Рисунок 8.4 – Виклик функції

8.2 Функції типу void

Функції типу **void** – ті, що не повертають значення – це певний різновид команд, реалізований особливими програмними операторами. Оператор **func()** (див. рис. 8.3) виконує функцію **void func()**, тобто передасть керування функції, поки не виконаються всі її оператори. Коли функція поверне керування в основну програму, тобто завершить свою роботу, основна програма продовжить своє виконання з тієї позиції, де розташовується наступний оператор за оператором **func()** .

На рисунку 8.5 наведено приклад організації та виклику функції для обчислення факторіалу числа, яка не повертатиме в головну програму жодного значення.

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

void faktorial(int);//прототип функції

int main()
```

```
{
    int number;//змінна для збереження введеного значення
    printf(" What factorial should be calculated? ");
    scanf_s("%d", &number);
    faktorial(number); //виклик функції обчислення факторіала
    return 0;
}
//визначення функції
void faktorial(int value) //заголовок функції
{
    int rezult = 1;
    for (int i = 1; i <= value; i++)
    {
        rezult *= i;
    }
    printf("%d! = %d", value, rezult);
}
```

Рисунок 8.5 – Приклад функції типу **`void`**

Якщо функція повертає значення типу, то її виклик слід організовувати так, щоб значення, яке повертається, не використовувалося. Тобто, таку функцію не використовують у правій частині виразу.

8.3 Функції, що повертають значення

Якщо функція має повертати певне значення, то вона повинна містити оператор повернення з певним значенням. Оператор **`return`** завершує виконання функції, в якій він викликається, при цьому результатом виклику такої функції є відповідне значення. При цьому в тілі такої функції необхідно реалізувати алгоритм обчислення значення-результату. Значенням може бути константа, змінна або вираз.

Оператор **`return`** переважно є останнім оператором тіла функції, але він може записаний у будь-якому місці тіла функції. Якщо алгоритм функції досить складний, то операторів **`return`** може бути декілька.

Синтаксис оголошення функції, що повертає значення будь-якого типу:

```
ім'я типу /*ім'я функції*/(< тип ім'я, …, <тип ім'я>)
{
      <оголошення ідентифікаторів змінних>
      <оператори>
      return <значення>;
}
```

Рисунок 8.6 – Синтаксис оголошення функції, що повертає значення

ім'я типу – тип значення, що повертається функцією, яке не може бути **void**;

<ім'я функції> – унікальний в межах програми ідентифікатор функції, за яким вона викликається;

<оголошення ідентифікаторів змінних> – оголошення ідентифікаторів, що використовуватимуться лише в межах функції;

<оператори> – оператори, які виконуються під час виклику функції, їх називають також **тілом функції**.

Функції, які повертають значення, по завершенню своєї роботи повертають результат, який має ***співпадати*** з ***типом даних в заголовку.***

Такі функції можуть повертати значення будь-якого типу даних.

На рисунку програму для обчислення факторіалу числа.

```
//Обчислення факторіала
#include <stdio.h>
#include <math.h>
//Прототип функції обчислення n!
int faktorial(int);
int main(){
    int number;//змінна для збереження значення n
    printf(" Which number's factorial should be calculated? ");
    scanf_s("%d", &number);
    int Rezult = faktorial(number); //запуск (виклик) функції
обчислення факторіала, number – фактичний параметр
    printf("Faktorial of %d is equal %d", number, Rezult);
    return 0;
}
//Визначення функції обчислення n!
int faktorial(int value) {//заголовок функції,
                          //value – формальний параметр
int rezult = 1;
    for (int i = 1; i <= value; i++)
        rezult *= i;
    return rezult;
}
```

Рисунок 8.7 – Обчислення факторіалу числа

Якщо функція повертає результат, тобто тип результату функції не є **void**, то її виклик можна присвоїти змінній того ж самого типу, що й сама функція, або вивести на екран за допомогою функції виведення на екран, наприклад так: **int Rezult = faktorial(number);** або **printf("Faktorial of %d is equal %d", number, faktorial(number));**

8.4 Передавання параметрів у функцію

Існує два способи передавання параметрів у функцію: **за значенням** і **за адресою**.

При передаванні за значенням в стек заносяться копії значень аргументів, і оператори функції працюють з цими копіями. Функція не має доступу до початкових значень параметрів, отже відсутня можливість їх змінити. При передаванні за адресою в стек заносяться копії адрес аргументів, а функція здійснює доступ до комірок пам’яті за цими адресами через операцію розіменування і може змінити значення аргументів.

```
#include <stdio.h>

void f(int i, int* j, int &k);

int main() {

	int i = 1, j = 2, k = 3;
	printf("i = %d, j = %d, k = %d\n", i, j, k);// Результат 1 2 3
	f(i, &j, k);
	printf("i = %d, j = %d, k = %d", i, j, k);//Результат 1 3 4
	return 0;
}

void f(int i, int* j, int& k) {
	i++; (*j)++; k++;
}
```

Рисунок 8.8 – Передавання параметрів у функцію

Перший параметр (**i**) передається за значенням. Його зміна в функції не впливає на вихідне значення. Другий параметр (**j**) передається за адресою за допомогою покажчика, при цьому для передавання у функцію адреси фактичного параметра використовується операція взяття адреси, а для отримання його значення в функції потрібна операція розіменування. Третій параметр (**k**) передається адресою за допомогою посилання.

Примітка. Передавання параметра у функцію за допомогою посилання можлива у C++. Для мови програмування C така операція неможлива – передавання має здійснюватись або за значенням або за покажчиком.

При передаванні за посиланням у функцію передається адреса зазначеного при виклику параметра, а всередині функції всі звернення до параметру неявно розіменовуються. Використання посилань замість покажчиків, поперше, покращує читабельність програми, позбавляючм від необхідності застосовувати операції одрежання адреси та розіменування; і, по-друге, не вимагає копіювання параметрів, що важливо при передаванні структур даних великого обсягу.

Розглянемо, яким чином можна передавати масив у функцію

Приклад. Написати програму, в якій генерується одновимірний масив випадкових чисел та обчислюється сума кубів елементів масиву

Дана програмна реалізація побудована таким чином: створюється динамічний масив **massive**, який ініціалізується випадковими елементами в діапазоні від 0 до 19. Потім здійснюється виклик функції **Sum(massive, N)** для обчислення суми кубів елементів масиву. Масив передається в функцію за адресою. Також, слід звернути увагу, на те, що в програмі реалізовано дві користувацькі функції. Перша **Cube(int x)** – обчислює та повертає куб числа. Друга функція **Sum(int* arr, int lenght)** – по-перше, викликає функцію для обчислення куба елементу масива **Cube(arr[i])**, куди передає

по черзі всі елементи масиву, а потім обчислює суму кубів цих елементів (див. рис. 8.9).

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
//Прототипи функцій
long int Cube(int);  // оголошення функції, яка повертатиме куб елементу
long int Sum(int* arr, int lenght); // передаємо до аргументів функції покажчик на масив та кількість його елементів

int main()
{
    int N;
    printf(«Enter number of elements in massive \n»);
    scanf_s(«%d», &N);
    int* massive = new int[N];
    printf(«Ramdom generated massive is …\n»);
    srand(time(NULL));
        //рандомна генерація елементів масиву
    for (int i = 0; i < N; i++) {
        massive[i] = rand() % 20; //кожен елемент масиву - випадкове число від 0 до 19
        printf(«%3d», massive[i]);
    }
    long int Suma = Sum(massive, N); //виклик функції
    printf(«\n Suma = %3d», Suma);
    return 0;
}

long int Cube(int x) //визначення функції, яка повертатиме куб числа
{
    long int y = pow(x, 3);
    return y;
}

long int Sum(int* arr, int lenght)//визначення функції, яка обчислюватиме суму кубів
{
    long int S = 0;
    for (int i = 0; i < lenght; i++) {
        S += Cube(arr[i]); //виклик функції Cube для піднесення до третього степеня кожного елементу масиву
    }
    return S;
}
```

Рисунок 8.9 – Передавання у функцію одновимірного масиву та обробка його елементів

Результат роботи програми представлено на рисунку 8.10

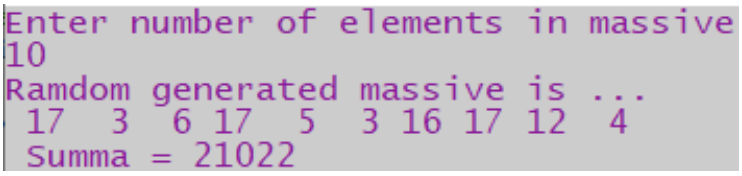


Рисунок 8.10 – Результат роботи програми

Приклад. Розробити програму для пошуку максимального елементу двовимірного масива.

В даній програмній реалізації слід звернути увагу на декілька моментів. По-перше створення динамічного двовимірного масиву **mmm**: спочатку виділяється пам'ять для стовпця вказівників на рядки матриці (допоміжний масив), далі – в циклі виділяється пам'ять для вказівників на рядки матриці (основний масив). Для обчислення максимуму матриці реалізовано функцію **int maxValueOfArray(int** a, int size1, int size2)**, яка приймає такі параметри: покажчик на масив, кількість рядків, кількість стовпців матриці. Через покажчик другого порядку можна передати масив у функцію тільки, якщо він динамічний. Виклик функції відбувається в основній програмі так: **maxValueOfArray(mmm, H, W)**, а оскільки ім'я масиву – це покажчик на його початок, то «&» не потрібен. Вивільнення пам'яті відбувається в оберненому порядку відповідно до виділення пам'яті під масив.

```
#include <stdlib.h>
#include <stdio.h>
int maxValueOfArray(int** a, int size1, int size2); //прототип
функції для пошуку максимального значення

int main()
{
	int H, W; //кількість рядків та стовпців відповідно
	scanf_s("%d %d", &H, &W);
	int** mmm = (int**)malloc(sizeof(int*) * H);//виділяємо
пам'ять для стовпця вказівників на рядки матриці
	for (int i = 0; i < H; i++) {
		mmm[i] = (int*)malloc(sizeof(int) * W);//виділяємо
пам'ять для вказівників на кожний рядок
	}
	//заповнення масиву
	for (int i = 0; i < H; i++) {
		for (int j = 0; j < W; j++) {
			mmm[i][j] = j + i;
			printf("%4d", mmm[i][j]);
		}
		printf("\n");
	}
	printf("\n");
	//виклик функції та передавання параметрів
	int MaximumOfArray = maxValueOfArray(mmm, H, W);
```

```
	printf("\nMaximal element of array is %d", MaximumOfArray);
	//вивільнення пам'яті
	for (unsigned int i = 0; i < H; i++) {
		free(mmm[i]);
	}
	free(mmm);
	return 0;
}
//Оголошення функції
int maxValueOfArray(int** a, int size1, int size2) {
	int max = a[0][0];
	for (int i = 0; i < size1; i++) {
		for (int j = 0; j < size2; j++) {
			if (a[i][j] > max)
				max = a[i][j];
		}
	}
	return max;
}
```

Рисунок 8.11 – Обчислення максимуму двовимірного масиву

Результат роботи програми представлено на рисунку 8.12

```
7 10
  0   1   2   3   4   5   6   7   8   9
  1   2   3   4   5   6   7   8   9  10
  2   3   4   5   6   7   8   9  10  11
  3   4   5   6   7   8   9  10  11  12
  4   5   6   7   8   9  10  11  12  13
  5   6   7   8   9  10  11  12  13  14
  6   7   8   9  10  11  12  13  14  15

Maximal element of array is 15
```

Рисунок 8.12 – Результат роботи програми

8.5 Рекурсивні функції

Рекурсія – це спосіб організації обчислювального процесу, при якому функція звертається сама до себе. Такі звернення називаються **рекурсивними викликами**, а функція, що містить рекурсивні виклики, – **рекурсивною**. Це означає, що для вирішення певної задачі потрібно серед допоміжних підзадач вирішити таку саму задачу, тільки з іншими значеннями параметрами.

Глибина рекурсії – максимальна кількість копій рекурсивної функції, що одночасно може знаходитись в пам’яті комп’ютера.

Щоб рекурсивний виклик працював коректно і не призвів до нескінченного циклу (що викликає помилку «переповнення стека» – Stack Overflow), він повинен завжди містити дві ключові частини:

1. Базовий випадок (Умова зупинки). Це найпростіший випадок задачі, який функція може вирішити **без виклику самої себе**. Основним призначенням базового випадку є забезпечення завершення рекурсії. Коли функція досягає

базового випадку, вона повертає результат, і процес починає «виринати» зі стека викликів.

2. Рекурсивний крок. Це частина, де функція **викликає сама себе**, але з **меншими або простішими** вхідними даними. Кожен рекурсивний виклик повинен наближати виконання програми до базового випадку. Якщо цього не відбувається, рекурсія стає нескінченною.

На рисунку 8.13 наведено структуру опису рекурсивних функцій.

```
if <умова завершення>
   <оператор>
else
   <виклик функції з іншими параметрами>
```

Рисунок 8.13 – Визначення рекурсивної функції

Кожний рекурсивний виклик створює нову копію змінних рекурсивної функції, при цьому стара копія змінних не знищується, а зберігається в стеку. Це єдиний випадок, коли під час виконання програми назві однієї змінної відповідають кілька її копій.

Цей процес відбувається в такій послідовності.

1. Резервування місця для формальних параметрів для запису копій фактичних параметрів.

2. При виклику функції в стек записується точка повернення, тобто адреса тої частини програми, де міститься виклик функції.

3. На початку тіла функції у стеку резервується місце для локальних (автоматичних змінних).

Стек пам’ятає історію рекурсивних викликів у вигляді послідовності (ланцюгів) таких фреймів. Програма у кожний конкретний момент працює з останнім викликом і з останнім фреймом. По завершенні рекурсії програма повертається до попередньої версії рекурсивної функції й до попереднього фрейму у стеку.

Від глибини рекурсії значною мірою залежить час виконання програми та об’єм потрібної стекової пам’яті. Значні витрати стекової пам’яті пов’язані з тим, що в рекурсивній підпрограмі, як правило, оголошується певна кількість локальних об’єктів: змінних, констант, вкладених підпрограм тощо.

Схему рекурсивних викликів наведено на рисунку:

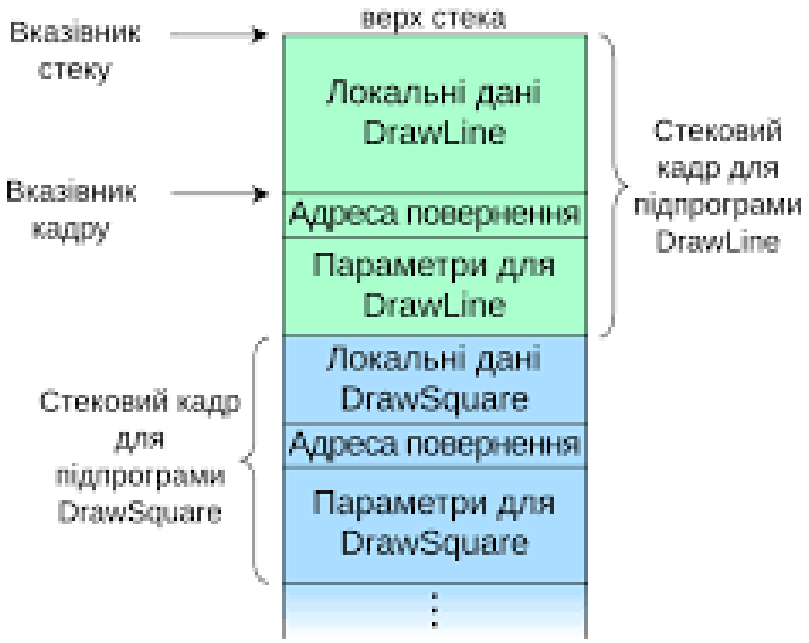


Рисунок 8.14 – Схема сегменту (фрейм) стеку виклика функцій та локальні змінні та інструкції по поверненню

Кожного разу, коли підпрограма викликається рекурсивно, виділяється стекова пам'ять для всіх її локальних об'єктів, і тому велика глибина рекурсії може призвести до нестачі стекової пам'яті. Якщо в тілі підпрограми (функції) здійснюється більше ніж один рекурсивний виклик, то надвеликими можуть стати не лише витрати стекової пам'яті, але і витрати часу. У цьому разі, можливо, перевагу варто надати нерекурсивному розв'язання задачі.

Приклад. Розробити рекурсивну функцію для обчислення факторіала.

Повне рекурсивне визначення факторіала має такий вигляд:

$$n! = \begin{cases} 1, \text{ якщо } n = 0 \\ (n-1)! * n \text{ якщо } n > 0 \end{cases}$$

Рекурентне співвідношення для обчислення факторіала позначається так: $n! = n * (n-1)!$

Умова зупинки рекурсії: якщо *n*=0 чи *n*=1, то факторіал дорівнює 1.

Тоді рекурсивна функція обчислення факторіала цілого числа має такий вид:

```
int RecFact(int n)
{
     if (n == 0 || n == 1) return 1;
     else return (n * RecFact(n - 1));
}
```

Рисунок 8.15 – Рекурсивна функція обчислення факторіала

Наведемо декілька коментарів до цієї реалізації. Нехай потрібно обчислити факторіал числа 3. При виклику функції **`RecFact(3)`** буде

викликана функція **RecFact(2)**, яка у свою чергу викличе **RecFact(1)**. Для останньої функції виконається умова зупинки рекурсії **(n==1)**, у функцію **RecFact(2)** повернеться значення **1** і обчислиться значення **2 (2!=2)**, яке у свою чергу повернеться до функції **RecFact(3)** і буде використане для обчислення значення **3!**. Схематично цей процес можна позанчити так:

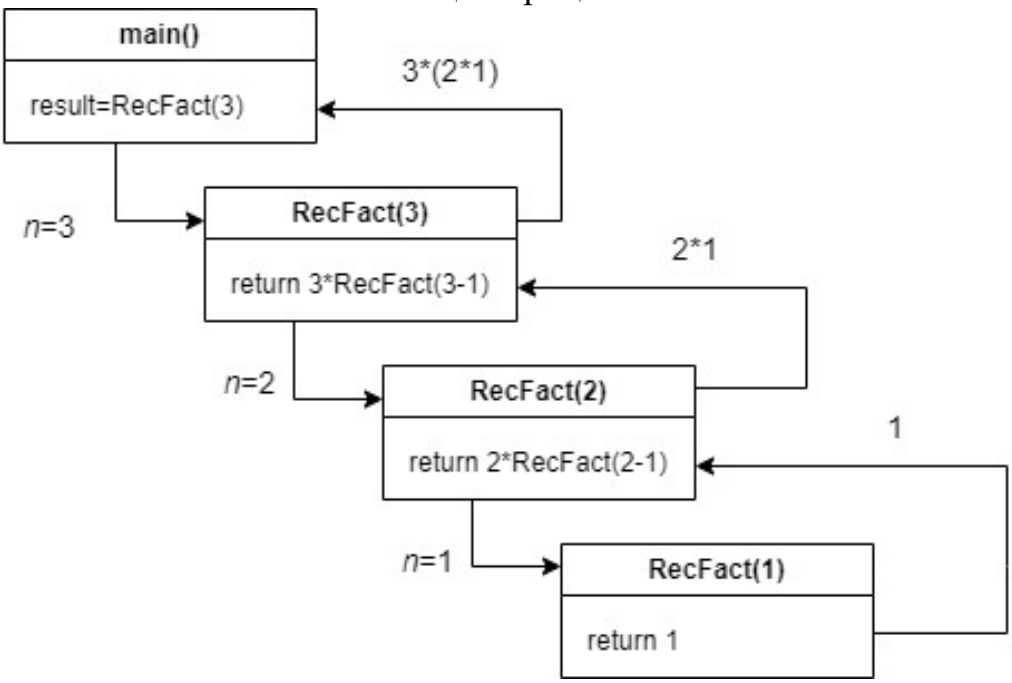


Рисунок 8.16 – Схема обчислення 3!

Тести для самоконтролю

1. Яка частина оголошення функції визначає тип даних, який функція повертає після свого виконання?

А. Тип значення, що повертається.

Б. Список параметрів.

В. Ім’я функції.

2. Для чого використовується ключове слово void в оголошенні функції?

А. Для позначення того, що функція може приймати будь-яку кількість параметрів.

Б. Для позначення того, що функція не повертає жодного значення.

В.Для позначення того, що функція використовує лише глобальні змінні.

3. Яке ключове слово використовується в тілі функції для негайного повернення результату (значення) та виходу з функції?

А. break.

Б. continue.

В. return.

4. Що таке рекурсивна функція?

А. Функція, яка завжди повертає значення типу void.

Б. Функція, яка викликає сама себе.

В. Функція, яка викликає іншу функцію.

5. При передаванні параметрів за значенням, що отримує функція?

А. Адресу змінної, що дозволяє змінити оригінал.

Б. Копію значення змінної, тому зміна всередині функції не впливає на оригінал.

В. Лише константи.

6. Як можна змусити функцію змінити значення оригінальної змінної, оголошеної поза функцією?

А. Передаючи змінну за значенням.

Б. Передаючи вказівник змінної.

В. Використовуючи ключове слово const.

7. Який елемент обов’язково повинна мати будь-яка рекурсивна функція для запобігання нескінченному циклу?

А. Блок else.

Б. Базова умова (умова виходу), яка зупиняє подальші рекурсивні виклики.

В. Аргумент, що завжди дорівнює нулю.

8. Яка з наступних функцій не повертає жодного значення?

А. int calculate_sum(int a, int b).

Б. double get_average(int data[]).

В. void print_message(char *msg).

9. Що таке прототип функції?

А. Перша змінна, оголошена всередині функції.

Б. Повна реалізація функції.

В. Рядок, який оголошує ім’я функції, тип, що повертається, і параметри, але без її тіла.

10. При передаванні масиву у функцію, як він фактично передається?

А. Використовуючи глобальні змінні.

Б. За посиланням (функції передається вказівник на перший елемент масиву).

В. За значенням (функція отримує повну копію всього масиву).

Практичні завдання

1. Напишіть функцію типу void, яка приймає вказівник на ціле число і збільшує значення цього числа утричі.

2. Напишіть рекурсивну функцію, яка обчислює суму всіх цілих чисел від 1 до заданого числа N.

Контрольні запитання

1. В чому різниця між механізмами передавання параметрів за значенням та за адресою?

2. В яких ситуаціях функція типу void є найбільш доречною?

3. Чому базова умова є критично важливою для коректної роботи рекурсії?

4. Що таке прототип функції?
5. Як відбувається передача масиву як параметра у функцію?

Тема 9. Файли.

План

1. Основні функції для роботи з файлами в С.
2. Загальна схема роботи з файлом.
3. Запис та читання інформації з файлу

9.1 Основні функції для роботи з файлами в С.

Основні функції файлового введення/виведення оголошено у заголовному файлі **`stdio.h`**. Серед іншого, в цьому ж файлі описано значення NULL, EOF тощо.

Таблиця 9.1 – Основні функції файлової системи С

Функція	Призначення
`fopen( )`	Відкрити файл
`fclose( )`	Закрити файл
`putc( )`	Записати символ у файл
`fputc( )`	Записати символ у файл
`getc( )`	Зчитати символ із файла
`fgetc( )`	Зчитати символ із файла
`fputs( )`	Записати рядок у файл
`fseek( )`	Встановити вказівник поточної позиції на певний байт файла
`ftell( )`	Повернути значення вказівника поточної позиції у файлі
`fprintf( )`	Друкувати у файл
`fscanf( )`	Зчитувати з файлу
`feof( )`	Повернути значення **`true`**, якщо досягнуто кінець файла
`ferror( )`	Повернути значення **`true`**, якщо трапилася помилка
`rename( )`	Перейменувати фізичний файл
`remove( )`	Знищити фізичний файл

Всі операції виконуються за допомогою **потоку** даних (stream), який є абстракцією, що пов'язує програму з зовнішнім файлом на диску.

9.2 Загальна схема роботи з файлом

Схема роботи з файлом складається з таких пунктів:

1. Оголосити **файлову змінну-вказівник** на структуру даних типу **FILE**. Наприклад, так:

```
FILE* myfile;
```

Рисунок 9.1 – Оголошення файлової змінної

Цей вказівник (**myfile**) є «ключем» до файлу. Він зберігає всю інформацію, необхідну для керування потоком: поточну позицію читання/запису, буфер даних, і режим доступу.

2. Відкрити файл, при цьому файл, розташований на диску, зв'язується з файловою змінною

```
#include < stdio.h >
FILE *fopen(const char *filename, const char *mode);
```

Рисунок 9.2 – Синтаксис команди для відкриття файлу

Безпосередньо у програмі відкриття файлу виглядатиме так: **myfile = fopen("hello.txt", "r")**, де **r** – параметр відкриття файлу, що означає відкриття для читання.

Основні режими доступу до файлу наведено в таблиці:

Таблиця 9.2 – Основні режими доступу

Режим	Призначення	Якщо файл не існує
"r"	Відкрити для читання даних.	Помилка (NULL)
"w"	Відкрити для запису.	Створює новий файл або перезаписує існуючий.
"a"	Відкрити для додавання даних у кінець.	Створює новий файл.
"r+"	Відкрити для читання та запису.	Помилка (NULL)
"w+"	Відкрити для читання та запису.	Перезаписує існуючий файл.

Функція **fopen()** відкриває файл і пов'язує його з потоком. Вона повертає вказівник типу **FILE *** або **NULL**, якщо файл не вдалося відкрити (наприклад, через відсутність файлу або недостатність прав доступу). Якщо файлу не існує – робота програми завершиться з помилкою.

Задля безпеки після виклику **fopen()** завжди необхідно перевіряти, чи не повернув він **NULL.**

```
FILE* data_file = fopen("data.txt", "r");

  if (data_file == NULL) {
      printf("Помилка: Не вдалося відкрити файл!\n");
      return 1;
  }
```

Рисунок 9.3 – Перевірка відкриття файлу

3. Виконати необхідні дії з файлом (опрацювання файла), наприклад зчитати з файла дані, надрукувати у файл дані, як продемонтсровано на рисунку 9.4.

```
fscanf(myfile, "%s %d", str, &a);
fprintf(myfile, "%s %d", str, a);
```

Рисунок 9.4 – Читання з та запис до файлу відповідно

4. Закрити файл, при цьому звільняються ресурси операційної системи, що використовуються для доступу до файла. Функція **`fclose()`** розриває зв'язок між програмою та файлом. Вона звільняє буфери, записує в файл усі незавершені дані та звільняє вказівник **`FILE *`**. **Закриття файлу є обов'язковим** для запобігання втраті даних та витоку пам'яті.

```
    fclose(myfile);
```

Рисунок 9.5 – Закриття файлу

9.3. Запис та читання інформації з файлу

Наведемо декілька прикладів читання та запису інформації у файл.

Приклад. Створити текстовий файл та записати в нього ціле число. Програмна реалізація матиме такий вид:

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

int main()
{
    int num;
    FILE* fptr;
    fopen_s(&fptr, "C:\\temp/program.txt", "a");
    if (fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }
    printf("Enter num: ");
    scanf_s("%d", &num);

    fprintf(fptr, "%d", num);
    fclose(fptr);

    return 0;
}
```

Рисунок 9.6 – Запис цілого числа у файл

Наведемо декілька коментарів до цієї програми. По-перше, дану програмну реалізацію виконано в середовищі MS VisualStudio, а тому для відкриття файлу використано функцію **`fopen_s`** з додатковим аргументом **`&fptr`**. Також, оскільки файл розташовано не в робочому каталозі, то для роботи з ним вказано повний шлях. Далі відбувається перевірка, чи знайдено файл з таким ім'ям за вказаною адресою. Якщо файл не знайдено, то

відбувається завершення програми. Якщо файл знайдено, то відбувається запит на введення цілого числа з клавіатури, а потім запис цього числа у файл. Після цього файл закривається.

Приклад. Прочитати інформацію, яка міститься в текстовому файлі. Програмна реалзіація матиме такий вигляд:

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

int main()
{
    int num;
    FILE* fptr;

    if ((fptr = fopen("C:\\temp/program.txt", "r")) == NULL) {
        printf("Error! opening file");
        exit(1);
    }

    fscanf (fptr, "%d", &num);

    printf("Value of n=%d", num);
    fclose(fptr);

    return 0;
}
```

Рисунок 9.7 – Зчитування інформації з файлу

Наведемо декілька коментарів до цієї програми. Для відкриття файлу використано функцію **`fopen`** з параметром **`r`**. Другий параметр – повний шлях до файлу. Після перевірки існування та можливості відкриття файлу відбувається зчитування інформації, далі – виведення на екран. Зкриття файлу реалізовано через функцію **`fclose`**.

Тести для самоконтролю

1. Який тип даних є обов’язковим для оголошення вказівника на файл у мові С?

А. int.
Б. FILE *.
В. char *.

2. Яка функція використовується для відкриття файлу та встановлення режиму доступу (читання/запис)?

А. fprintf().
Б. fclose().
В.fopen().

3. Який режим доступу слід використовувати, якщо потрібно записати дані у файл, а якщо файл не існує, створити його?

А. «w».

Б. «r».

В. «a».

4. Що повертає функція fopen(), якщо файл не вдалося відкрити (наприклад, через відсутність дозволів або помилковий шлях)?

А. Від’ємне число.

Б. Нуль.

В. Значення NULL.

5. Яка функція є дзеркальним відображенням printf() і використовується для форматованого запису даних у файл?

А. fputs().

Б. fprintf().

В. fgetc().

6. Яка функція є обов’язковою для виклику після завершення роботи з файлом?

А. rewind().

Б. fclose().

В. free().

7. Який режим доступу слід використовувати, якщо потрібно дописати дані до кінця вже існуючого файлу?

А. «a» (додавання).

Б. «r+» (читання і запис).

В. «w» (запис)

8. Яка функція є дзеркальним відображенням scanf() і використовується для форматованого читання даних із файлу?

А. fgets().

Б. fscanf()

В. fread().

9. Який режим роботи з файлами є кращим для зберігання структурованих даних (наприклад, масивів або структур) без перетворення у текст?

А. Текстовий режим («r», «w», «a»).

Б. Режим лише для читання («r»).

В. Бінарний режим («rb», «wb», «ab»).

10. Який загальний алгоритм роботи з файлами є коректним?

А. Відкриття - Читання/Запис - Закриття - Обробка помилок.

Б. Відкриття - Перевірка на NULL - Читання/Запис - Закриття.

В. Читання - Обробка - Відкриття - Закриття.

Практичні завдання

1. Напишіть програму для обчислення коренів квадратного рівняння. Коефіцієнти зчитати з файла, корені записати у файл.

2. Створіть програму на С, яка здійснює запис структурованих даних у текстовий файл, а потім зчитує ці дані для базового аналізу.

Контрольні запитання

1. Чому перевірка вказівника на файл на значення NULL одразу після виклику fopen() є найважливішим етапом у роботі з файлами?
2. В якому разі випадкове використання режиму «w» призведе до незворотної втрати даних?
3. В яких ситуаціях вибір бінарного режиму є більш ефективним та точним для зберігання структур даних або масивів, ніж текстовий режим?
4. Що таке буферизація вводу/виводу?
5. Які проблеми із сумісністю можуть виникнути, якщо при передаванні бінарного файлу у текстовому режимі?

Додаток А

Перелік зарезервованих слів мови програмування С

Ключове слово	Призначення
auto	Специфікатор класу зберігання
break	Переривання циклу або switch
case	Мітка у блоці switch
char	Тип даних для символів (1 байт)
const	Специфікатор константності змінної
continue	Перехід до наступної ітерації циклу
default	Значення за замовчуванням у блоці switch
do	Початок циклу з постумовою
double	Тип даних для чисел із плаваючою комою подвійної точності
else	Альтернативний блок виконання для if
enum	Оголошення перелічуваного типу (enumeration)
extern	Оголошення змінної або функції, визначеної в іншому файлі
float	Тип даних для чисел із плаваючою комою одинарної точності
for	Початок циклу з лічильником
goto	Безумовний перехід
if	Умовний оператор
int	Тип даних для цілих чисел
long	Модифікатор для збільшення діапазону цілих чисел
register	Специфікатор класу зберігання (запит на розміщення у регістрі процесора)
return	Завершення функції та повернення значення
short	Модифікатор для зменшення діапазону цілих чисел
signed	Модифікатор для цілих чисел зі знаком
sizeof	Оператор, що повертає розмір об'єкта в байтах
static	Специфікатор класу зберігання (локальна видимість)
struct	Оголошення структури
switch	Оператор вибору (багатоваріантне розгалуження)
typedef	Оголошення нового імені для існуючого типу
union	Оголошення об'єднання (union)

Ключове слово	Призначення
unsigned	Модифікатор для цілих чисел без знака
void	Тип для позначення відсутності значення (наприклад, функція нічого не повертає)
volatile	Модифікатор, що вказує на можливу зміну змінної поза програмою
while	Початок циклу з передумовою

Рекомендована література

1. Козак Л.І., Костюк І.В., Стасевич С.П. Основи програмування : навчальний посібник. Львів : «Новий Світ-2000», 2024. 328 с.
2. Ковалюк Т.В. Алгоритмізація та програмування : підручник. Львів : «Магнолія 2006», 2022. 400 с.
3. Мартін Роберт С. Чистий код : створення і рефакторинг за допомогою Agile. Харків : Ранок. 2020, 448 с.
4. Шпурик В.В., Олєнєва К.М. Основи програмування. Професійний підхід. Київ : КПІ ім. Ігоря Сікорського, 2024. 106 с.
5. Пекарський Б.Г. Основи програмування: навчальний посібник. Київ : Кондор, 2024. 364 с.
6. Preschern C. Fluent C: Principles, Practices, and Patterns. O'Reilly Media, 2024. 277 p.
7. Richard M Reese. Understanding and Using C Pointers: Core Techniques for Memory Management. O'Reilly Media, 2020. 225 p.
8. Gustedt J. Modern C. Manning Publications Co, 2024. 437 p.

Використана література

1. Козак Л.І., Костюк І.В., Стасевич С.П. Основи програмування : навчальний посібник. Львів : «Новий Світ-2000», 2024. 328 с.
2. Ковалюк Т.В. Алгоритмізація та програмування : підручник. Львів : «Магнолія 2006», 2022. 400 с.
3. Ришковець Ю.В., Висоцька В.А. Алгоритмізація та програмування. Частина 1: навчальний посібник. Львів : «Новий світ - 2000», 2021. 337 с.
4. Мартін Роберт С. Чистий код : створення і рефакторинг за допомогою Agile. Харків : Ранок. 2020, 448 с.
5. Пекарський Б.Г. Основи програмування: навчальний посібник. Київ : Кондор, 2024. 364 с.
6. Richard M Reese. Understanding and Using C Pointers: Core Techniques for Memory Management. O'Reilly Media, 2020. 225 p.
7. Gustedt J. Modern C. Manning Publications Co, 2024. 437 p.

Інформаційні джерела

1. Основи програмування CS50. URL : https://prometheus.org.ua/prometheus-free/osnovy-programuvannya-cs50-2019/ (дата звернення 06.12.2025)

2. Керниган Б., Рітчі Д. Мова програмування С. 2020. URL : https://web.archive.org/web/20130523201358/http://void.net.ua/The_C_Programming_Language.html (дата звернення 06.12.2025)

3. Уроки програмування на C++. URL : https://acode.com.ua/uroki-po-cpp/ (дата звернення 06.12.2025)

4. C\C++ Programming Style Guidelines. URL : https://geosoft.no/development/cppstyle.html (дата звернення 06.12.2025)

5. Interactive C Course URL : https://www.programiz.com/c-programming/c-file-input-output (дата звернення 06.12.2025)

Навчальне видання
(*українською мовою*)

Кривохата Анастасія Григорівна
Кудін Олексій Володимирович

ОСНОВИ ПРОГРАМУВАННЯ

Навчальний посібник
для здобувачів ступеня вищої освіти бакалавра
спеціальності «Інженерія програмного забезпечення»
освітньо-професійної програми «Програмна інженерія»

Рецензент *О.С. Пшенична*
Відповідальний за випуск *А.О. Лісняк*
Коректор *А.Г. Кривохата*