О.А. Кобилін
В.А. Любченко
Д.О. Руденко
І.Ю. Кириченко

Основи програмування мовою C/C++

Харків – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ
УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

О.А. Кобилін, В.А. Любченко,
Д.О. Руденко, І.Ю. Кириченко

Основи програмування мовою C/C++

Електронне видання

Харків – 2025

УДК 004.43
О75

Рекомендовано до друку рішенням Вченої ради
Харківського національного університету радіоелектроніки
(протокол № 2/3 від 04.02.2025 р.)

О75 **Кобилін О.А., Любченко В.А., Руденко Д.О., Кириченко І.Ю.**
Основи програмування мовою С++: Навчальний посібник / [Електронний ресурс] О.А.Кобилін, В.А. Любченко, Д.О. Руденко, І.Ю. Кириченко. — Електронне видання. – Харків: ХНУРЕ, 2025. – 307 с. – pdf 3,81 Mb.

ISBN 978-966-659-418-4

Посібник містить матеріал, що вивчається дисциплінами «Інформатика», «Програмування», «Алгоритмічні мови та програмування» тощо. В його розділах надано необхідні теоретичні відомості з програмування мовою С++, наведено схеми алгоритмів та приклади програм, а також вправи для самостійного розв'язання. Видання охоплює головні аспекти мови С++ і є практичною настановою їхнього використання як для різних програмних середовищ, так і для режиму з інтерфейсом користувача системи С++.

Навчальний посібник призначений для студентів молодших курсів вищих навчальних закладів, але він буде корисний і для більш досвідчених програмістів та викладачів.

ISBN 978-966-659-418-4
DOI: 10.30837/978-966-659-418-4

ЗМІСТ

Комп'ютерні програми пишуться не тільки для машин,
але й для людей, які їх використовують.
Єдиний засіб вивчення нової мови програмування —
написання на ній програм.
Б. Керніган

ПЕРЕДМОВА

Широке розповсюдження інформаційних технологій в усі сфери людської діяльності висуває підвищені вимоги до комп'ютерної підготовки фахівців у вищих навчальних закладах. Здійснення цих вимог неможливе без досконалого освоєння сукупності процесів, що пов'язані з розробкою програм та їхньою реалізацією, тобто програмуванням. І тому програмування та розв'язання задач на комп'ютері — дисципліни, що входять у перелік найбільш перспективних, які займають важливе місце в системі підготовки спеціалістів різного профілю. Обчислювальні машини були створені взагалі для розв'язання складних інженерних задач, і, незважаючи на значне розширення сфери їхнього застосування, цей напрямок використання залишається головним.

Як мова програмування вибрана найбільш популярна алгоритмічна мова C/C++. Аналіз літератури показав, що не всі видання надають перевагу глибокому вивченню практичного застосування тих чи інших конструкцій цієї мови. Тому головне місце в посібнику займає вивчення основних питань програмування мовою C/C++ з їхнім безпосереднім практичним використанням.

Навчальний посібник призначений для студентів молодших курсів закладів вищої освіти, які володіють азами програмування з самого початку, але він буде корисний також і більш досвідченим програмістам та викладачам.

Це видання не претендує на повноту викладення матеріалу щодо мови програмування C/C++, бо для цього існують довідники, документація та контекстна допомога. Автори ставили за мету створення практичної настанови з використання мови C/C++, яка надавала б уявлення про основні можливості цієї мови та способи її практичного застосування.

Структура посібника є, на наш погляд, зручною та мотивованою, тому що чітко відповідає прийнятому у закладі вищої освіти принципу навчання: виклад теоретичного матеріалу на лекції чергується

з наступним закріпленням його на практичних заняттях. Основну увагу приділено надбанню практичних навичок розв'язання прикладів при засвоєнні окремих розділів вибраної алгоритмічної мови.

Тим, хто робить перші кроки в програмуванні, необхідно ознайомитися з основами алгоритмізації. З цією метою треба звернути особливу увагу на перший розділ посібника, де надаються початкові уявлення про дані, алгоритми, створення блок-схем. Наведені типові прийоми складання алгоритмів з їхнім детальним поясненням сприятимуть розвитку логічного мислення. Після засвоєння цього «ядра знань» можна перейти безпосередньо до програмування.

Далі надаються загальні відомості про мову програмування C++, її основні конструкції, необхідну довідкову інформацію та визначення, що достатні для розв'язання задач. У цих розділах поступово розглядаються такі основні питання мови, як типи даних, змінні та константи, операції та оператори, масиви та покажчики, символьні та рядкові дані, дані типу структура, функції, файли, класи тощо. Багаторічний досвід авторів у викладанні дисципліни «Програмування» свідчить про те, що одночасне вивчення головних особливостей конструкцій будь-якої алгоритмічної мови та опрацювання відповідних текстів програм сприяє кращому розумінню матеріалу. Досягти достатньої глибини в розкритті кожного з розглянутих питань вдалося завдяки великій кількості розв'язаних прикладів. Програмна реалізація цих прикладів може бути здійснена за допомогою будь-яких компіляторів мови C++, що сумісні із стандартом мови.

У розділі «Вправи» посібник містить систематизовані та тематично розміщені завдання. Сподіваємося, що сумлінні читачі розглядатимуть приклади та виконуватимуть на комп'ютері запропоновані вправи та, врешті-решт, це стане для них цікавим і корисним захопленням.

Для раціонального відбору матеріалу автори скористалися великим практичним досвідом, набутим ними у процесі викладання на кафедрі інформатики Харківського національного університету радіоелектроніки під час підготовки здобувачів різних спеціальностей.

Уважний читач знайде на сторінках цього посібника багато корисного, теоретичні викладки та приклади практичних реалізацій програм допоможуть йому дати відповіді на запитання, що були викладені менш глибоко в інших настановах та виданнях.

1 ОСНОВНІ ПОНЯТТЯ АЛГОРИТМІЗАЦІЇ

1.1 Етапи розв'язання задач на комп'ютері

Незважаючи на велику різноманітність програм, у самому процесі їхнього створення можна знайти щось узагальнююче. Розглянемо основні етапи підготовки та розв'язання задач за допомогою комп'ютера.

Постановка задачі. Розв'язання будь-якої задачі починається з її постановки, викладеної мовою чітко визначених математичних понять. При цьому необхідно добре уявити суть поставленої задачі, необхідні початкові дані та інформацію, що вважається результатами розв'язання.

Побудова математичної моделі. Не завжди умова сформульованої задачі містить у собі готову математичну формулу, яку можна застосувати для розробки алгоритму задачі, і не завжди розв'язок задачі вдається отримати в явному вигляді, що пов'язує вихідні дані та результат. Для цього створюється інформаційна математична модель об'єкта, і чим достовірніше вона відображає реальні сторони об'єкта, тим точніші отримані результати. Тут особливо важлива однотипність методів розв'язання задач.

Розробка алгоритму. Створення алгоритму, тобто послідовності вказівок для розв'язання задачі, відбувається на основі побудованої математичної моделі.

З метою знаходження способу розв'язання поставленої задачі можуть бути застосовані вже відомі методи, проведена їхня оцінка, аналіз, відбір або розроблені нові методи. Під час створення складних алгоритмів застосовується метод покрокової розробки, сутність якого полягає в тому, що алгоритм розробляється «зверху донизу». Такий підхід дозволяє розбити алгоритм на окремі частини, кожна з яких розв'язує свою самостійну підзадачу, та далі об'єднати ці підзадачі в єдине ціле.

Складання програми. Для вирішення задачі за допомогою комп'ютера алгоритм має бути записаний мовою програмування. Процес розробки програми потребує гарного знання вибраної мови програмування та може здійснюватися теж за принципом «зверху донизу», що дозволяє отримати добре структуровану програму, читання та розуміння якої значно полегшене.

Компіляція програми. Переведення програми на машинну мову здійснюється за допомогою спеціальних програм — компіляторів. Однією з функцій компілятора є перевірка у програмі синтаксичних помилок і, за їхньої відсутності, побудова об'єктного модуля.

Компонування програми здійснюється компонувальником (редактором зв'язків), який формує виконавчий модуль програми. На цьому етапі відбувається підключення бібліотек, з'єднання окремих модулів, тобто розв'язання зовнішніх посилань.

Налагодження програми. Окрім синтаксичних помилок, програма може мати помилки іншого типу — змістовні, логічні. Вони з'являються під час помилкового трактування умови поставленої задачі через недосконалість математичної моделі або недоліки у побудованому алгоритмі. Процес налагодження програми полягає в підготовці системи тестів, які містять набір вихідних даних, що мають відомий результат. Якщо для всіх тестів результати роботи програми збіглися з розрахунками, то можна вважати, що логічних помилок немає.

Експлуатація програми. Програма, що має відповідну документацію, може бути тиражована і запропонована іншим користувачам.

1.2 Поняття алгоритму

Алгоритм є фундаментальним поняттям інформатики [3, 9—11, 16]. Відомий середньоазіатський мудрець, вчений, філософ і математик Мухаммед бен Муса аль-Хорезмі у IX ст. детально розробив правила чотирьох арифметичних дій (їх можна назвати алгоритмами арифметичних дій). При перекладі його наукових трактатів вперше з'явився термін «**алгоритм**» (*аль-Хорезмі — Algorithmi*).

Алгоритм — це наперед заданий чіткий опис скінченної послідовності вказівок, виконання яких дозволяє отримати правильний (бажаний) розв'язок задачі (результат).

Алгоритм повинен відповідати певним вимогам і мати такі **властивості**:

- ***визначеність*** (детермінованість) — алгоритм має бути чітким і однозначним, кожна команда не повинна допускати довільності тлумачення, кожний крок алгоритму має бути точно визначеним;
- ***масовість*** — алгоритм повинен бути по можливості універсальним, розрахованим на розв'язання однотипних задач з різними вихідними даними;
- ***дискретність*** — визначений алгоритмом обчислювальний процес повинен мати дискретний (перервний) характер, тобто бути послідовністю окремих завершених кроків — команд або дій;
- ***результативність*** — кожна дія має приводити до певного результату;
- ***формальність*** — будь-який виконавець, діючи за алгоритмом, може реалізувати поставлене завдання;
- ***скінченність*** — розв'язок задачі з використанням алгоритму має бути одержаним за скінченну кількість кроків.

Запис алгоритмів здійснюється під час їхньої розробки та подання. Алгоритм є певною інструкцією для виконавця, яку можна задати різними способами — словами, формулами, послідовністю обчислюваних операцій чи логічних дій тощо. На практиці застосовують різні способи запису алгоритмів у текстовій та графічній формі.

Велика кількість реальних завдань досить складна, тому між словесним описом дій і програмою на алгоритмічній мові виконується проміжний етап — **побудова схеми алгоритму**. Цей етап становить найбільш наочний спосіб зображення алгоритмів у вигляді графічних схем. Схема є нібито начерком структури програми та «путівником» за вже готовою програмою. При графічному способі зображення алгоритмів необхідно виконувати вимоги стандарту. В основі цього способу лежить поняття символу дії, який зображується окремою геометричною фігурою (табл. 1.1).

Під час створення схеми алгоритму блоки із записаними в них командами з'єднуються між собою стрілками для визначення черговості виконання дій алгоритму. Для запису команд всередині блоків використовується природна мова з елементами математичної символіки. Розробити алгоритм — це розбити задачу на етапи (більш прості задачі), що послідовно виконуються. При цьому чітко зазначається як зміст кожного етапу, так і порядок їхнього виконання.

Основні типи алгоритмів — це три типові (базові) структури, які можна використовувати для розробки алгоритмів різного ступеня складності: лінійні, розгалуження та цикли.

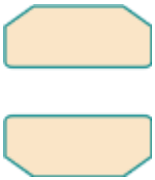
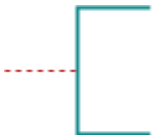
Лінійним алгоритмом називається такий алгоритм, в якому лінійні команди виконуються послідовно одна за одною.

Розгалуженим алгоритмом називається алгоритм, що містить хоча б одну умову, в результаті перевірки якої здійснюється перехід до одного з можливих кроків. Процес розгалуження організується за допомогою логічного операторного блока.

Циклічний алгоритм містить повторення кілька разів з новими вихідними даними певної послідовності команд, що утворює тіло циклу. У циклі повинна існувати змінна (параметр циклу), яка при кожному наступному виконанні тіла циклу змінює своє значення і визначає кількість повторень.

Дані та їхні структури важливо чітко уявляти під час розробки алгоритмів [3]. При цьому враховуються не тільки властивості, але і структурні особливості об'єктів (даних), над якими виконуються перетворення під час розв'язання задачі. Кожний об'єкт повинен мати унікальне ***ім'я*** ***(ідентифікатор)***, що вибирається розроблювачем алгоритму та складається, як правило, з послідовності літер і цифр. Комп'ютер виділяє конкретному об'єкту визначене місце в пам'яті, де зберігатиметься його значення.

Таблиця 1.1 – Опис основних символів схем алгоритмів

	Блок, що зображує початок чи кінець алгоритму; всередині блока треба написати «початок» або «кінець»
	Блок введення-виведення даних зображується паралелограмом. Текст усередині блока конкретизує операцію, що виконується, і містить слово «введення» або «виведення», а також імена змінних, які необхідно ввести чи вивести
	Арифметичний операторний блок зображується прямокутником. Використовується для позначення дій, що задають або змінюють значення величин. Найчастіше всередині блока записують вираз з використанням математичних символів
	Логічний операторний блок (блок прийняття рішення) зображується ромбом. Використовується для вибору одного з двох можливих напрямків виконання алгоритму (стрілки з позначками «так» і «ні»). Всередині блока записується умова вибору, перехід по стрілці з позначкою «так» відбувається, коли умова виконується, а перехід по стрілці з позначкою «ні» — у протилежному випадку
	Блоки початку та кінця циклу. Всередині блока початку циклу записують діапазон та крок зміни параметра циклу, а у блоці кінця циклу — вказується ім’я параметра циклу
	Блок виклику підпрограми (функції, модуля), тобто допоміжних алгоритмів, які визначені автономно. Всередині блока записується, наприклад, ім’я функції, яка викликається, та список фактичних параметрів
	Символ, що використовується для зв’язку елемента схеми з коментарем.
	Символ-з’єднувач застосовується для обриву лінії схеми та продовження її у другому місці

Елемент даних, що має фіксоване значення, яке під час розв'язання задачі не змінюється, називають ***константою***.

Поряд з константами широко використовуються ***змінні***, тобто величини, що мають ідентифікатор і значення, що може змінюватися залежно від виконаних дій. У випадку, коли змінна **x** приймає значення, наприклад, в інтервалі від **0** до **1**, тобто $\mathbf{x} \in \mathbf{[0; 1]}$, і змінюється з постійним кроком $\mathbf{h_x} = \mathbf{0.1}$, її значення можна представити послідовністю із 11 чисел:

0; 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 0.7; 0.8; 0.9; 1.

Ця послідовність може зберігатися в пам'яті комп'ютера двома способами:

- всі значення змінної **x** зберігаються по черзі в одній комірці пам'яті та обчислюються самим комп'ютером за заданим законом змінення кроку $\mathbf{h_x}$ згідно з межами ($\mathbf{x} \in \mathbf{[0; 1]}$);
- у пам'яті виділяється стільки комірок, скільки значень приймає змінна, тобто існує масив з 11 послідовних комірок, де для переходу від одного значення до іншого необхідно змінювати номер комірки.

Таким чином, залежно від способу зберігання розрізняють прості змінні та змінні з індексами (масиви).

Масив — структура даних, що визначена ім'ям (ідентифікатором) і є однорідною, фіксованою за розміром і упорядкованою за номерами (індексами) сукупністю елементів. Індекс першого елемента дорівнює 0. У загальному випадку елементами масиву можуть бути будь-які однотипні дані. Найчастіше використовуються одновимірні масиви (вектори) та двовимірні масиви (матриці), наприклад: $\mathbf{x_i}$ **(i = 0...n-1)**, **n = 11** або $\mathbf{A_{ij}}$**(i = 0...n-1, j = 0...m-1)**, **n = 4**, **m = 6**. Такі масиви розрізняють за кількістю індексів, які потрібно вказувати для звернення до їхніх елементів.

1.3 Приклади найпростіших типових алгоритмів

Розглянемо деякі типові прийоми алгоритмізації, які на практиці використовуються або у вигляді окремих алгоритмів, або входять до складу більш складних алгоритмів. При цьому треба зважати на те, що та ж сама задача може бути розв'язана різними способами.

Основними методами побудови алгоритмів є розробка розгалужених алгоритмів, організація простих і вкладених циклів, обчислення суми, добутку, кількості, обробка одновимірних та багатовимірних масивів: визначення максимального та мінімального значень масивів, сортування масивів за зростанням чи за спаданням тощо.

Реальні задачі мають суперпозицію або композицію цих типових засобів. Практично будь-який алгоритм містить один чи декілька циклів, тобто ділянок, що виконуються багаторазово. У простому циклі змінюється один параметр від заданого початкового до кінцевого значення з постійним кроком (залежно від вимог задачі параметр циклу може збільшуватися або зменшуватися). При кожному значенні параметра виконуються оператори, що знаходяться всередині циклу, тобто тіло циклу. Вихід з циклу здійснюється після досягнення параметром циклу свого кінцевого значення.

Приклад 1.1.

Обчислити значення функції $\mathbf{y} = \mathbf{ax^2} - \mathbf{sin(x)}$, якщо $\mathbf{x} \in \mathbf{[-1;2]}$; $\mathbf{h_x} = \mathbf{0.5}$; $\mathbf{a} = \mathbf{10.5}$, та знайти кількість додатних значень функції.

У цьому прикладі **проста змінна x** є аргументом функції, який змінюється з кроком $\mathbf{h_x}$.

На рисунку 1.1 наведено схему алгоритму розв'язання поставленої задачі, яка включає:

- блок введення значень коефіцієнта **a** та кроку зміни аргументу $\mathbf{h_x}$;
- блоки присвоювання початкових значень змінній **k** ($\mathbf{k} = \mathbf{0}$) та аргументу **x** ($\mathbf{x} = -\mathbf{1}$); змінна **k** використовується для підрахунку кількості додатних значень функції;
- реалізацію багаторазового обчислення значень функції **y** при різних значеннях аргументу **x**, яке здійснюється циклічною ділянкою алгоритму, що виконує такі дії, як обчислення значення функції, виведення значень **x** і **y**;
- розгалуження, що міститься в тілі циклу за параметром **x**, реалізоване блоком перевірки умови $\mathbf{y > 0}$. У випадку, коли **y** додатний, відбувається ***підрахунок кількості додатних значень функції*** $\mathbf{k} = \mathbf{k} + \mathbf{1}$;
- якщо додатний елемент функції підраховано або **y** від'ємний, значення аргументу **x** збільшується на крок $\mathbf{x} = \mathbf{x} + \mathbf{h_x}$;
- логічний блок, що керує циклом, містить перевірку нерівності $\mathbf{x} \le \mathbf{2}$ і, якщо **x** перевищить значення **2**, забезпечує вихід з циклу.

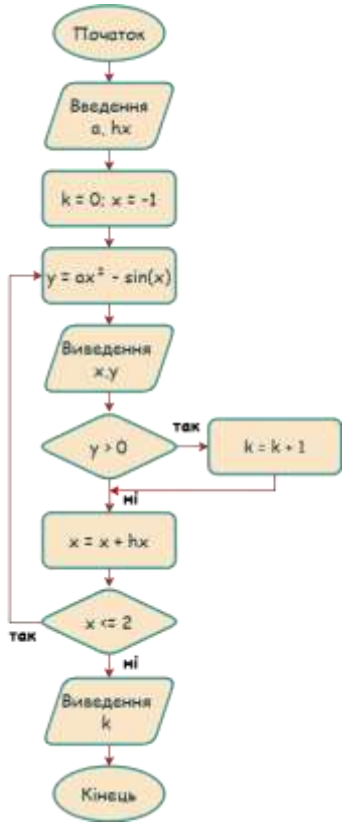


Рисунок 1.1 – Схема алгоритму прикладу 1.1

Приклад 1.2.

Скласти алгоритм обчислення значень функції **y**, аргументи якої розташовано в масиві $\mathbf{x_i(i = 0...n-1)}$, **n**=**20** і **a**=0.75; **b**=1.19; **c**=2.5.

$$\mathbf{y} = \begin{cases} \mathbf{ax_i + b\cos(x_i) + c}, & \text{якщо } \mathbf{x_i \le 0,5}; \\ \mathbf{bx_i^2 + c\sin(2x_i)}, & \text{для всіх інших значень } \mathbf{x_i}. \end{cases}$$

Знайти добуток значень функції, що перевищують величину **1.75** та суму всіх інших значень функції.

Схему алгоритму розв'язання даної задачі наведено на рис. 1.2.

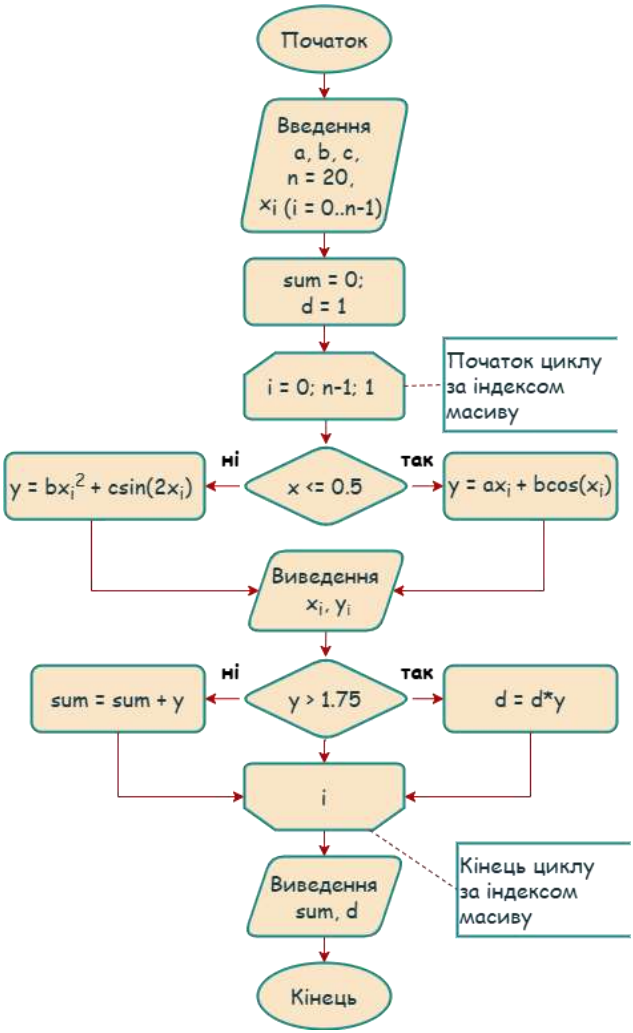


Рисунок 1.2 – Схема алгоритму прикладу 1.2

Алгоритм розв'язання складається з таких етапів:

– спочатку відбувається послідовне введення в пам'ять аргументів функції — елементів одновимірного масиву $\mathbf{x_i}$ **(i = = 0…n-1), n = 20**, тобто блок введення передбачає для цього використання циклу за індексною змінною **i**;

– для організації доступу до потрібного елемента масиву та виконання необхідних обчислень використовується новий цикл за параметром **i**;

– згідно з умовою задачі в алгоритмі передбачено два розгалуження;

– у тілі циклу перша ділянка розгалуження починається з перевірки умови $\mathbf{x_i \le 0.5}$, оскільки область визначення функції поділяється на дві частини, в кожній з яких значення **y** обчислюється за окремою формулою;

– всередині циклу відбувається виведення поточних значень **y** та $\mathbf{x_i}$;

– друга ділянка розгалуження починається з перевірки умови **y > 1.75** та здійснює обчислення добутку та суми;

– якщо умова **y > 1.75** не виконується, реалізується обчислення суми: початкове значення змінної суми **sum** дорівнює нулю (**sum = 0**) і ця інструкція розташована ще до початку циклу, а далі у циклі багаторазово виконується накопичення суми **sum = sum + y**;

– у випадку виконання умови **y > 1.75**, аналогічно здійснюється створення добутку **d**: початкове значення добутку дорівнює одиниці (**d = 1**), а подальше значення обчислюється як **d = d*y**;

– після завершення циклу за параметром **i**, тобто коли всі аргументи $\mathbf{x_i}$ вичерпано, здійснюється виведення одержаних значень суми **sum** і добутку **d**.

Приклад 1.3.

За один перегляд масиву $\mathbf{c_i}$ (**i = 0...n-1**), **n = 15** визначити значення та положення максимального та мінімального його елементів і поміняти їх місцями.

Схему алгоритму розв'язання задачі зображено на рисунку 1.3.

Для знаходження максимального та мінімального елементів заданого одновимірного масиву скористаємося ***типовим прийомом — введенням допоміжної*** (***робочої***) ***змінної.*** Максимальний елемент позначимо через **max**, а його позицію — **imax**. Відповідно для мінімального елемента та його позиції використовуються змінні **min** та **imin**. Перед переглядом масиву як в **max**, так і в **min** записується перший елемент масиву, тобто елемент, розташований в позиції **i = 0** (**max** = $\mathbf{c_0}$ та **min** = $\mathbf{c_0}$), а його номер (**0**) запам'ятовується змінними **imax** та **imin** (**imax = 0**; **imin = 0**).

У процесі порівняння, наприклад, **max** з $\mathbf{c_1}$, якщо змінна **max** буде більша за елемент $\mathbf{c_1}$, то значення **max** зберігається, у протилежному випадку **max** набуде значення $\mathbf{c_1}$. Отже, в будь-якому

разі змінна **max** матиме максимальне значення серед c_0 та c_1. Аналогічно аналізуються значення $c_2, \ldots, c_{n-1}$.

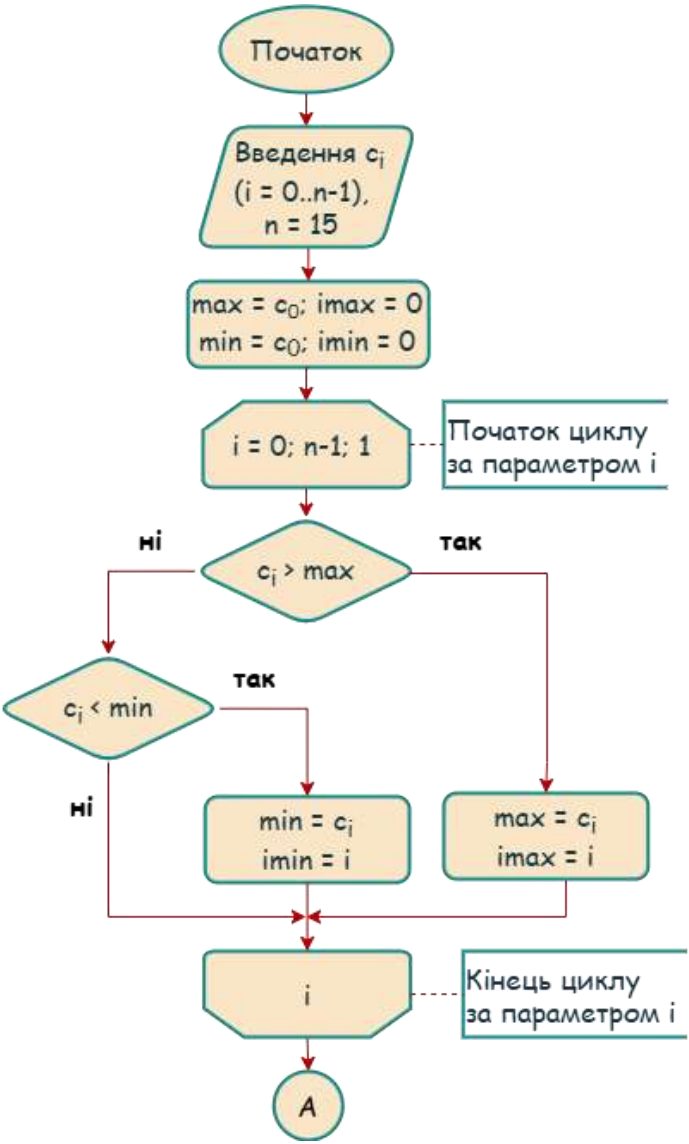


Рисунок 1.3, а – Схема алгоритму прикладу 1.3

Рисунок 1.3, б – Схема алгоритму прикладу 1.3 (продовження)

Таким чином, здійснюючи пошук максимального елемента серед сукупності чисел, необхідно послідовно їх переглянути та порівняти між собою. Для цього дії над елементами масиву необхідно реалізувати циклом, параметром якого є індекс **i** елемента масиву. В тілі циклу зміна значень параметрів **max** та **imax** (тобто **max** = **c[i]; imax** = **i;**) відбувається у випадку, коли при перегляді зустрічається елемент $\mathbf{c_i}$, більший за **max** ($\mathbf{c_i}$>**max**). Значення **min** та **imin** змінюються на нові, якщо $\mathbf{c_i}$ < **min**.

Після закінчення циклу отримаємо значення та номери максимального та мінімального елементів. Для перестановки цих елементів місцями достатньо виконати дві дії, а саме: $\mathbf{c_{imax}}$ = **min** та $\mathbf{c_{imin}}$ = **max**.

Приклад 1.4.

Упорядкувати за зростанням масив $\mathbf{x_i}$ **(i = 0…n-1)**, **n**=**12**, з використанням обмінного сортування (за методом «бульбашки»).

Сортування — це упорядкування масиву за будь-якою ознакою. Існують різні методи сортування, серед них обмінне сортування (метод «бульбашки»)— найбільш простий, але й найменш ефективний метод сортування, його доцільно застосовувати для сортування невеликих масивів. Простота алгоритму (рис. 1.4) та програмної реалізації роблять метод досить популярним.

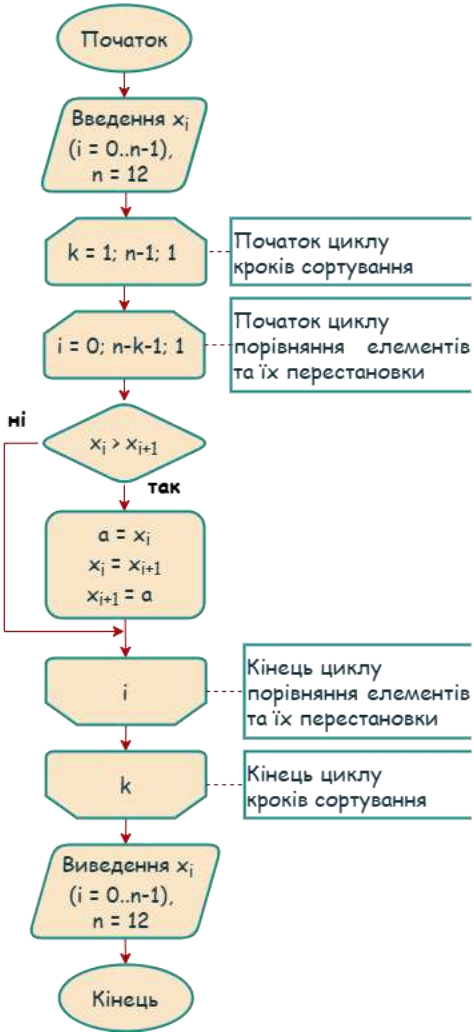


Рисунок 1.4 — Схема алгоритму прикладу 1.4

Приклад 1.5.

Задана матриця $\mathbf{B_{ij}}$ (**i** = **0…n-1**, **j** = **0…m-1**) **n** = **4**, **m = 5**. Необхідно для кожного рядка матриці знайти кількість від'ємних елементів та записати їх в одновимірний масив.

Головна умова правильного розв'язання задач з матрицями полягає в розумінні порядку змінювання індексів її елементів. Перший індекс (наприклад, **i**) завжди визначає номер рядка, другий (наприклад, **j**) — номер стовпця.

Алгоритм розв'язання поставленої задачі (рис. 1.5) виконується так:

- спочатку елементи масиву $\mathbf{B_{ij}}$ (**i** = **0…3**, **j** = **0…4**) послідовно вводяться в пам'ять комп'ютера, тобто блок введення передбачає використання двох циклів — «за рядками» (за параметром **i**) та «за стовпцями» (за параметром **j**). **Двовимірні масиви вводяться «за рядками»**, тобто цикл за параметром **i** є зовнішнім, а цикл за параметром **j** — внутрішнім;
- стосовно організації циклічного процесу блок виведення матриці реалізується подібно до блока введення;
- оскільки підраховується кількість від'ємних елементів для кожного рядка матриці, то зовнішній цикл передбачений за параметром **i**;
- ***підрахунок кількості від'ємних елементів*** виконує параметр **k** (його початкове значення **k** = **0**);
- для здійснення процесу підрахунку кількості від'ємних елементів рядка необхідно перебирати всі його елементи, тобто використати цикл за параметром **j**, що буде внутрішнім відносно циклу за параметром **i**; для організації лічильника **k** необхідно скористатися типовим прийомом алгоритмізації (приклад 1.1), тобто **k** = **k** + **1**;
- для запам'ятовування результатів аналізу кожного рядка ($\mathbf{rob_i}$ = **k**) необхідно скористатися ***типовим прийомом алгоритмізації — формуванням робочого масиву*** $\mathbf{rob_i}$ **(i = 0…n-1)**, виведення елементів якого здійснюється після перегляду всіх рядків матриці;
- ***у процесі організації вкладених циклів*** рекомендується дотримуватися такого правила: ***параметр циклу***, ***що визначений останнім***, ***повинний мінятися першим*** (так, параметр **j** визначається після **i**, тому в першу чергу змінюється параметр **j**).

Доцільно зауважити, що у розглянутому прикладі здійснюється перегляд матриці «за рядками». При перегляді матриці «за стовпцями» порядок зміни індексів протилежний, тобто для кожного **j** індекс **i** змінюється потрібну кількість разів.

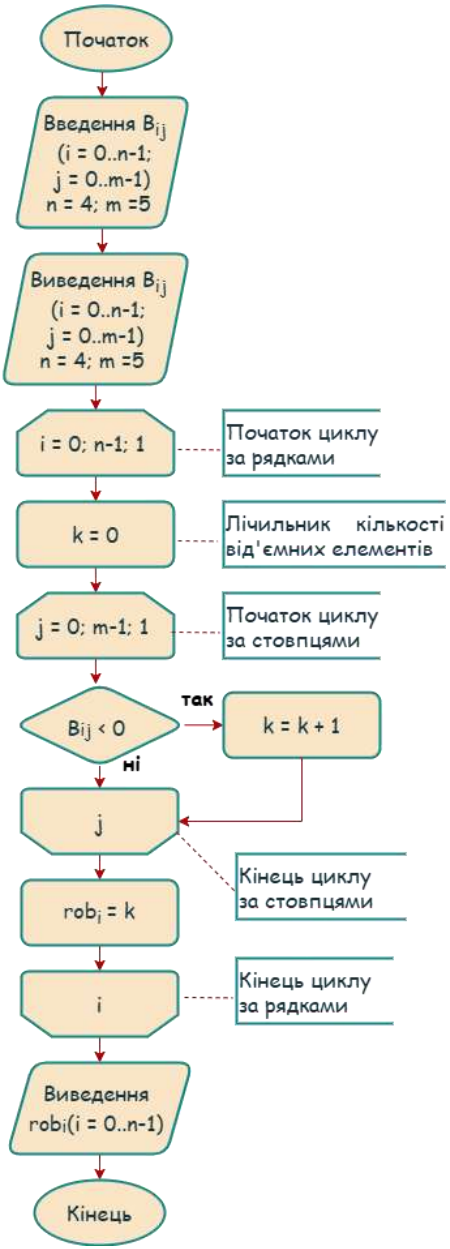


Рисунок 1.5 – Схема алгоритму прикладу 1.5

Приклад 1.6.

Парні елементи заданої матриці $\mathbf{a_{ij}}$ (**i** = **0…n-1**, **j** = **0…m-1**), **n** = **3**, **m** = **4** переписати до масиву **b**, а непарні — до масиву **c**.

Схему алгоритму розв'язання поставленої задачі зображено на рисунку 1.6.

Організація введення-виведення елементів заданої матриці $\mathbf{a_{ij}}$ (**i** = **0…n-1**, **j** = **0 … m-1**), **n** = **3**, **m** = **4**, здійснюється аналогічно до розглянутої у ***прикладі 1.5***.

Для формування двох одновимірних масивів, в які заноситимуться відповідно парні та непарні елементи, здійснюється перегляд усіх елементів вихідної матриці. З цією метою реалізуються два вкладених цикли — «за рядками» (за параметром **i**) та «за стовпцями» (за параметром **j**).

Алгоритм використовує два лічильники: **kc** та **kn** відповідно для підрахунку кількості парних і непарних елементів (початкові значення **kc** = **0** і **kn** = **0**). Зрозуміло, що змінні **kc** і **kn** одночасно становлять індекси масивів, які створюються. Організація лічильників **kc** і **kn** здійснюється за типовим прийомом алгоритмізації (приклад 1.1 і приклад 1.5).

Після перевірки умови на парність відбувається запис елементів матриці $\mathbf{a_{ij}}$ у відповідні масиви: $\mathbf{b_{kc}} = \mathbf{a_{ij}}$ та $\mathbf{c_{kn}} = \mathbf{a_{ij}}$. Етап формування масивів парних елементів $\mathbf{b_i}$ (**i** = **0…kc-1**) та не парних елементів $\mathbf{c_i}$ (**i** = **0…kn-1**) завершується після виконання вкладених циклів за параметрами **i** та **j**.

Виведення одержаних масивів виконується послідовно та передбачає використання циклу за індексною змінною **i**. З цією метою можна було б застосувати іншу змінну, тобто змінну з другим ім'ям. Але в подібних випадках краще використовувати змінні з програми, які до потрібного етапу її виконання закінчили своє функціонування. У даному прикладі параметр **i** виконав свої функції у циклі визначення парних та непарних елементів матриці і тому може брати участь у процесі виведення елементів масивів.

Підводячи підсумок, зазначимо, що у розділі наведено реалізації типових структур алгоритмів, а саме: лінійних, розгалужень та циклів. У процесі розробки алгоритму в першу чергу звертається увага на спосіб завдання початкових даних. Розглянуті приклади ілюструють два головних способи завдання даних:

- у вигляді простої змінної, яка змінює своє значення від початкової до кінцевої величини з деяким кроком;
- у вигляді масиву чисел.

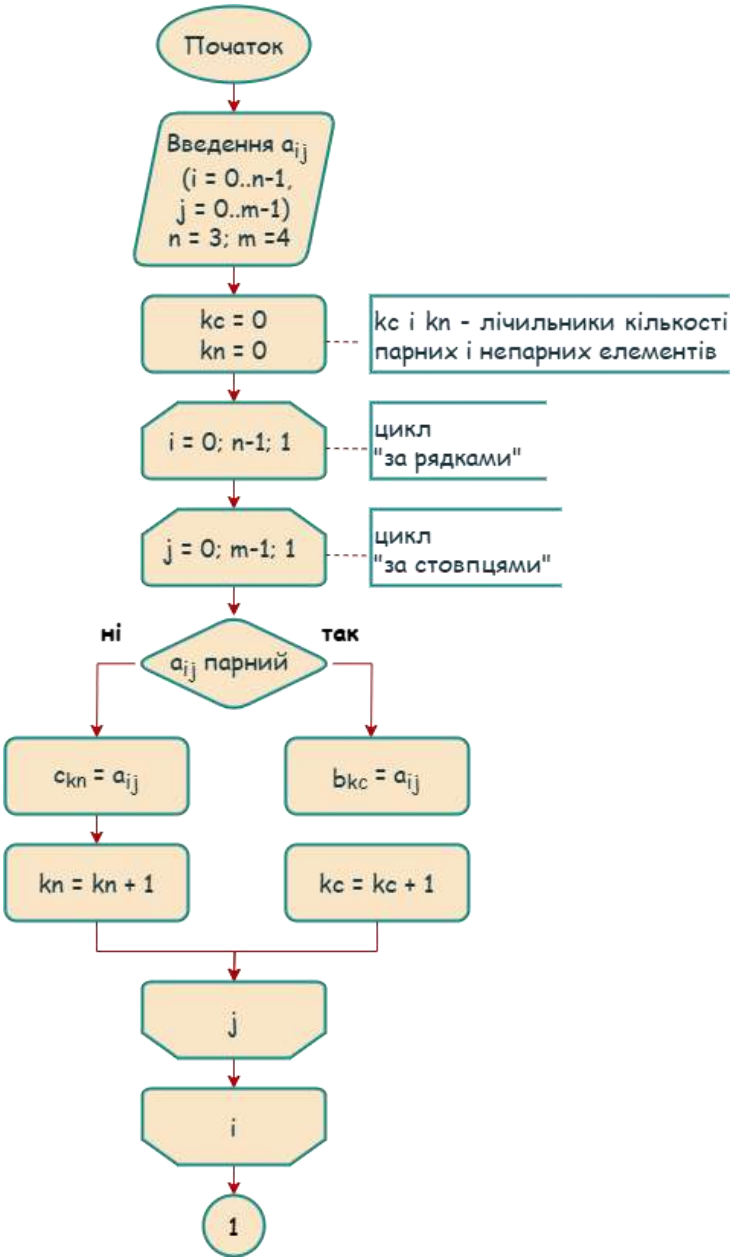


Рисунок 1.6, а – Схема алгоритму прикладу 1.6

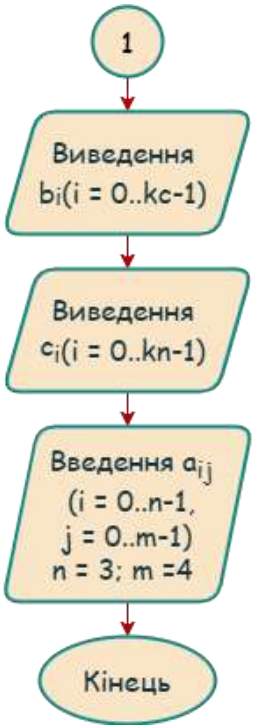


Рисунок 1.6, б – Схема алгоритму прикладу 1.6 (продовження)

Необхідно зауважити, що *у межах посібника схеми алгоритмів наведено з урахуванням важливої особливості мови С++: під час роботи з масивами нумерація елементів починається з нуля* (*0*), *а не з одиниці* (*1*).

1.4 Запитання та завдання

1. Назвіть та охарактеризуйте основні етапи розв'язання задач на комп'ютері.
2. Що таке алгоритм? У чому полягає суть побудови алгоритмів? Поясніть важливість тестування програми під час її налагодження.
3. Назвіть та охарактеризуйте основні властивості алгоритмів.
4. У чому полягають особливості схематичного зображення алгоритму?
5. Назвіть та охарактеризуйте три базові структури алгоритмів.

6. Поясніть на прикладах сутність розгалужень та лінійних структур алгоритмів.

7. Що таке цикл? Наведіть приклади простих та вкладених циклів.

8. Коли виникає необхідність введення проміжних даних?

9. У чому відмінність між виконанням обчислень суми, кількості та добутку?

10. Охарактеризуйте, як розрізняють прості змінні та масиви.

11. Поясніть особливості використання вкладених циклів під час обробки двовимірних масивів.

***Завдання*.**

Набути практичних навичок складання алгоритмів під час розробки їх для завдань розділів вправ «Циклічні обчислювальні процеси» та «Одновимірні та двовимірні масиви».

2 СКЛАД МОВИ С/С++

2.1 Стислий екскурс в історію

Пізнавши історію появи мов С та С++, легше зрозуміти концепції, що лежать в їхній основі, а також відповісти на запитання, чому протягом вже не одного десятиріччя мова С залишається популярною серед програмістів, а її більш молодша «родичка» С++ не поступається їй у популярності.

У 1972 році співробітник фірми Bell Laboratories Деніс Рітчі створив нову алгоритмічну мову — С. В її основу було закладено багато особливостей мови Assembler. Мова С є універсальною, придатною для розв'язання будь-якого типу задач, хоча спочатку була задумана як мова системного програмування (у 1973 році на мові С Деніс Рітчі реалізував операційну систему Unix).

У середині 1980-х років данський програміст Б'ярн Страуструп розробив мову програмування, яка спочатку називалася «С з класами». Вона стала основою для сучасної мови програмування С++.

Страуструп поєднав ефективність і низькорівневий доступ до апаратних ресурсів, характерні для мови С, з можливостями об'єктно-орієнтованого програмування (ООП), які він запозичив із мови Simula 67. Основною ідеєю було створення мови, яка б дозволяла ефективно працювати з великими й складними програмними системами.

У 1983 році назву змінили на С++, де «++» символізує операцію інкремента в С, натякаючи на еволюцію мови. Уже до кінця 1980-х років С++ став широко популярним і використовувався для розробки різноманітного програмного забезпечення, включаючи операційні системи, ігри, програми для наукових досліджень і багато іншого.

Мову С++ можна розглядати як надмножину мови С, бо вона зберігає усі можливості, що надає мова С, і доповнює їх засобами об'єктно-орієнтованого програмування. С++ є універсальною алгоритмічною мовою, яка використовується для розробки системних та складних прикладних програм. Це не тільки найпоширеніша мова програмування, але й мова спілкування програмістів, оскільки більшість програм алгоритмів написані мовою С++.

2.2 Алфавіт, лексеми, синтаксис мови

У природній мові спілкування виділяють 4 основні елементи: символ, слово, словосполучення та речення. Подібні елементи існують

і в алгоритмічній мові, тільки слова називаються **лексемами**, словосполучення називаються **виразами**, а речення — **операторами**. Лексеми створюються із символів, вирази — із лексем та символів, оператори — із символів, виразів і лексем [1, 6, 7, 12—14].

Алфавіт мови С++ включає:

- великі **(A–Z)** і малі **(a–z)** літери латинського алфавіту та символ підкреслення **(_)**;
- арабські цифри від **0** до **9**;
- знаки арифметичних дій **+**, **-**, *****, **/**, **%**, **++**, **--**;
- знаки побітових операцій <<, **>>**, **&**, **|**, **~**, **^**;
- знаки відношень <, <=, ==, !=, **>**, **>**=;
- знаки логічних операцій **&&**, **||**, **!**;
- розділові знаки **,**, **;**, **:**, пробіл;
- спеціальні знаки **.**, =, **->**, **?**, ****, **$**, **#**, **`**, **``**;
- символи дужок **(**, **)**, **[**, **]**, **{**, **}**.

Інші символи, а також літери кирилиці не використовуються для побудови базових елементів мови або для їхнього розділу, але вони можуть застосовуватися у символьних константах та коментарях.

Лексеми, тобто базові елементи мови з певним самостійним значенням, складаються із символів алфавіту. До них належать ідентифікатори, ключові слова, знаки операцій, константи, роздільники (дужки, крапка, кома, символи пробілу). Межі лексем визначаються іншими лексемами-роздільниками або знаками операцій.

Ідентифікатором, тобто ім'ям програмного об'єкта, називається будь-яка послідовність літер латинського алфавіту, цифр і символу підкреслення за умови, що першою стоїть літера або символ підкреслення, а не цифра.

Існує два різновиди ідентифікаторів:

- *стандартні*, наприклад, імена всіх вбудованих у мову функцій;
- *користувальницькі*.

Характерно, що мова С++ чутлива до регістру літер, тому компілятор розпізнає великі та малі літери латинського алфавіту як різні символи. Це дає можливість створювати ідентифікатори, що однаково читаються, але відрізняються написом одного або декількох символів. Наприклад, ідентифікатори «*Name*», «*name*» і «*NAME*» вважаються різними.

Ідентифікатори можуть мати будь-яку довжину, але значимими є не більше 31 символу від початку ідентифікатора, а в деяких компіляторах це обмеження ще більш суворе (не більше 8 символів). Імена програмних об'єктів створюються на етапі оголошення даних, після цього їх можна використовувати в різних операторах програми.

Можна надати декілька порад щодо вибору ідентифікатора:

– рекомендується не жалкувати часу на створення ідентифікаторів переважно за їхнім змістовим призначенням;

– ім'я програмного об'єкта повинно легко розпізнаватися і, бажано, не мати символів, які можна переплутати між собою;

– ідентифікатор не має збігатися з ключовими словами, а також з іменами стандартних об'єктів мови С++;

– для розділу частин імені можна використовувати символ підкреслення;

– всередині ідентифікатора не можна розміщувати символи пробілу.

Ключовими (***службовими***) ***словами*** називають ряд зарезервованих ідентифікаторів, що вживаються для побудови конструкцій мови та мають фіксоване значення. За смисловим навантаженням службові слова поділяються на такі основні групи:

– специфікатори типів — **char**, **int**, **long**, **typedef**, **short**, **float**, **double**, **enum**, **struct**, **union**, **signed**, **unsigned**, **void**;

– квалифікатори типів — **const** і **volatile**;

– класи пам'яті — **auto**, **extern**, **register**, **static**;

– для побудови операторів — **for**, **while**, **do**, **if**, **else**, **switch**, **case, continue**, **goto**, **break**, **return**, **default**, **sizeof**.

В таблиці 2.1 наведено список основних ключових слів С++.

Таблиця 2.1 – Ключові слова С/С++

asm	delete	goto	register	throw
auto	do	if	return	try
break	double	inline	short	typedef
case	else	int	signed	typename
catch	enum	long	sizeof	union
char	explicit	new	static	unsigned
class	extern	operator	struct	virtual
const	float	private	switch	void
continue	for	protected	template	volatile
default	friend	public	this	while

Як роздільники лексем застосовуються такі символи: пробіл, табуляція, символ нового рядка, коментар. Між будь-якими двома лексемами допускається довільна кількість символів-роздільників. Крім того, деякі лексеми («*****», «**+**», «**,**», «», «**(**», «**->**» тощо) самі є роздільниками та відділяти їх від інших лексем символами-роздільниками необов'язково. Між будь-якими двома лексемами допускається довільна кількість символів-роздільників.

2.3 Структура програми

Основними частинами типової структури програми на С++ є такі:

- директиви препроцесорної обробки;
- опис зовнішніх змінних (вихідних даних і результатів) та функцій;
- функції програми;
- головна функція програми **main()**, що має вигляд:

```
int main()
{
        опис змінних; виконавчі оператори;
return 0;  //повертаємо значення операційній системі
}
```

У загальному випадку програма складається з декількох функцій, що не перетинаються (тобто «вкладення» однієї функції в іншу неприпустиме). Перед функціями та між ними можуть бути присутні оголошення об'єктів даних і оператори препроцесорної обробки. Функції користувача, які викликаються у головній функції *main()*, необхідно обов'язково описати до їхнього використання. Наведемо приклад запису фрагмента простої програми:

```
#include <iostream>    // підключення бібліотеки введення/виведення
#include <stdlib.h>     /* підключення стандартної бібліотеки,
                        яка включає в себе функцію system()  */
int main()     //головна фукнція
{
  std::cout<<"Hello, world!"<<std::endl;  /* виведення повідомлення
                                          на екран */
  system("PAUSE");      /* затримка екрану (виконання команди
                        операційної системи) */

return 0;       //повертаємо значення операційній системі
}
```

Після виконання такої програми на екрані з'явиться повідомлення:

```
Hello, world!
```

Коментарі необхідні для пояснень призначення тих чи інших частин програми та їхній текст завжди ігнорується компілятором. Мова С++ використовує два різновиди коментарів:

- ***// текст — однорядковий коментар***, який починається з двох символів «/» («коса риска») і закінчується символом переходу на новий рядок;

– ***/*текст*/*** *—* ***багаторядковий коментар***, що розташовується між символами-дужками «/*» і «*/».

Багаторядкові коментарі не можуть бути вкладеними один в одному, а однорядкові коментарі можна вкладати в багаторядкові коментарі. Багаторядкові коментарі доцільно застосовувати для тимчасового виключення блоків при налагодженні програми.

Наведемо кілька порад стосовно раціонального складання коментарів:

– коментарі повинні бути добре складеними реченнями, мати правильну пунктуацію та містити тільки потрібну для супроводу інформацію;

– пробіл — один з найбільш ефективних коментарів, що значно поліпшує розуміння програми;

– штрихові лінії коментаря або порожні рядки застосовуються для поділу функцій та інших логічно завершених фрагментів програм.

Директива препроцесора **#include <iostream>** забезпечує підключення до програми засобів зв'язку зі стандартними потоками введення-виведення даних. Ці засоби знаходяться у заголовному файлі **iostream**, де **i** (***input***) — введення, **o** (***output***) — виведення, **stream** — потік, **h** (***head***) — заголовок.

При створенні програми враховують такі основні вимоги:

– усі використані константи, змінні, функції та нестандартні типи повинні бути оголошеними (описаними) до їхнього першого використання, ці оголошення можна розміщувати в будь-якому місці програми;

– кожний оператор мови закінчується символом «**;**»;

– фігурні дужки («**{**» та «**}**») виділяють складений оператор і все, що подано між такими дужками, синтаксично сприймається як один оператор;

– вкладені блоки повинні мати відступ у 3 – 4 символи, при цьому блоки одного рівня вкладеності необхідно вирівняти за вертикаллю.

2.4 Основні етапи створення executable в С/С++ (компіляція програми у загальному сенсі)

Назва етапу	Опис	Вхідні файли	Вихідні файли	Примітка
Препроцесорна обробка (для С і для С++)	Виконуються директиви препроцесора (#include, #define,….)	file1.h …. fileN.h file1.cpp (.c) …. fileN.cpp (.c)	file1.cpp (.c) …. fileN.cpp (.c) *Змінені *.cpp за правилами препроцесора *.h файли будуть вставлені в *.cpp*	
Розбір шаблонів (тільки для С++)	Шаблони (template) замінюються на конкретну реалізацію відповідно до типу	file1.cpp (.c) …. fileN.cpp (.c) *(файли з шаблонами)*	file1.cpp (.c) …. fileN.cpp (.c) *(файли без шаблонів)*	Якщо в початкових файлах шаблони відсутні, цей етап пропускається.
Компіляція	Перевірка синтаксису та створення об'єктних файлів *.cpp (.c) транслюється в *.obj (.o)	file1.cpp (.c) …. fileN.cpp (.c)	file1.obj (.o) …. fileN. obj (.o)	Компіляція успішна тільки у випадку відсутності синтаксичних помилок
Лінковка (компоновка)	Усі об'єктні файли «збираються» (лінкуються) в один файл, який запускається (executable)	file1.obj (.o) …. fileN. obj (.o)	file.exe	

2.5 Запитання та завдання

1. З чого складається алфавіт мови С++?
2. Охарактеризуйте на прикладах поняття «лексема».
3. Що таке ідентифікатор і які існують вимоги до його створення?
4. Які службові слова використовує мова С++?
5. Охарактеризуйте застосування символів-роздільників.
6. Які коментарі використовує мова С++? Охарактеризуйте поради щодо їхнього створення, наведіть приклади.

3 ДАНІ ТА ОПЕРАЦІЇ

3.1 Типи даних

Обробка даних різного типу є головною метою будь-якої програми. Кожне з даних характеризується класом пам'яті, ім'ям, типом і значенням. Імена дозволяють ідентифікувати дані, тобто відрізняти їх між собою. Програміст обирає тип кожної величини, що використовується для подання реальних об'єктів. Тип задає множину можливих значень даних і способи їхнього зберігання, перетворення та використання.

Обов'язкове оголошення типу даних дозволяє компілятору робити перевірку допустимості різних конструкцій програми.

Усі типи даних мови С++ можна розділити на **основні** (**базові**) та **складені**. Основні типи визначені для представлення цілих, дійсних, символьних і логічних даних. На основі цих типів вводиться опис складених типів, до яких належать масиви, перелік, функції, структури, посилання, покажчики, об'єднання та класи.

Основні типи даних (табл. 3.1) часто називають арифметичними, тому що їх можна використовувати в арифметичних операціях. Для опису основних типів мови С++ використовують такі службові слова:

- **int** (цілий);
- **char** (символьний);
- **bool** (логічний);
- **float** (дійсний);
- **double** (дійсний з подвійною точністю);
- **void** (порожній, не має значення).

Типи **int**, **char**, **bool** називають ***цілими***, а типи **float** і **double** — ***дійсними з плаваючою крапкою***. Код, що формує компілятор для обробки цілих величин, відрізняється від коду для величин з плаваючою крапкою.

Для уточнення внутрішнього подання та діапазону значень стандартних типів мова С++ використовує чотири специфікатори типу:

- **short** (короткий);
- **long** (довгий);
- **signed** (знаковий);
- **unsigned** (беззнаковий).

В таблиці 3.1 наведено діапазони значень та розміри основних типів даних (для 16-розрядного та 32-розрядного процесорів). Розмір однакового типу даних може відрізнятися на комп'ютерах різних

платформ, а також може залежати від застосованої операційної системи. Тому при оголошенні тієї чи іншої змінної потрібно чітко уявляти, скільки байт вона займатиме в пам'яті комп'ютера, щоб запобігти проблемам, пов'язаних з переповненням і неправильною інтерпретацією даних. Діапазони кожного з типів (табл. 3.1) повинні бути перевірені для конкретного комп'ютера.

Таблиця 3.1 – Базові типи даних (платформа Intel)

Тип	Розмір, байт	Значення
bool	1	true або false
unsigned short int	2	від 0 до 65 535
shortint	2	від -32 768 до 32 767
unsigned long int	4	від 0 до 4 294 967 295
long int	4	від -2 147 483 648 до 2 147 483 647
int (32 розряди)	4	від-2 147 483 648 до 2 147 483 647
unsigned int (16 розрядів)	2	від 0 до 65 535
unsigned int (32 розряди)	4	від 0 до 4 294 967 295
char	1	від 0 до 255 або від -128 до 127 залежно від компілятора та налаштувань компіляції
float	4	від ±1.18e-38 до ±3.4e38
double	8	від ±2.23e-308 до ±1.8e308
long double	8/12/16	від ±3.36e-4932 до ±1.18e+4932

3.2 Змінні

Кожна програма потребує виконання різноманітних обчислень, для здійснення яких використовуються **вирази**, що складаються з операндів, знаків операцій та дужок. Операнди задають дані для обчислень, а операції задають дії, які необхідно виконати над цими даними. Операнд є, у свою чергу, виразом, що в окремому випадку може бути константою або змінною.

Змінна — це іменована область пам'яті, в якій зберігаються дані визначеного типу. Змінна має ім'я, розмір та інші атрибути, такі як видимість, час існування тощо. Ім'я змінної служить для звернення до області пам'яті, у якій зберігається її значення. ***Перед використанням будь-яка змінна має бути описана*** *(або* ***оголошена****)*, при цьому для неї резервується деяка область пам'яті, розмір якої залежить від конкретного типу змінної. ***Під час виконання програми змінна може приймати різні значення***.

Наведемо загальний вигляд опису змінних:

```
[const] тип ім'я [= ініціювання або (ініціювання)]; ,
```

де **ініціювання** — тут це присвоювання змінній в ході опису початкового значення, яке записується зі знаком рівності «= **значення**» або в круглих дужках: (**значення**). (у посібнику при описі синтаксису об'єктів програмування необов'язкові частини синтаксичних конструкцій мови подано у квадратних дужках «**[ ]**»); модифікатор **const** вказує, що змінна не може змінювати своє значення, у цьому випадку її називають **типізованою** (**іменованою**) **константою** або просто **константою**; Зазначимо, що ***константа має бути ініційована під час опису***. Один оператор може містити опис декількох змінних одного типу, розділяючи їх комами, наприклад:

```
const int n = -2, m = 7, k = 444; //ініціювання констант n, m, k
                                  //цілого типу
float h = 0.175, d(-3.5), sum;    //опис дійсних змінних h, d, sum з
                                  // ініціюванням h, d
char sf = 't', st[] = "Learn to walk before you run";
                                  //ініціювання символьних змінних
```

Якщо **тип** значення, що ініціюється, не збігається з типом змінної, то виконуються перетворення типу. Кожна змінна повинна мати своє **ім'я**, причому в одному блоці не може бути двох змінних з однаковим ім'ям.

Опис змінної, крім типу, явно або за замовчуванням задає її область дії. Клас пам'яті та область дії залежать не тільки від опису, але і від місця їхнього розміщення в тексті програми.

Областю дії ідентифікатора змінної є частина програми, в якій його можна використовувати для доступу до зв'язаної з ним області пам'яті. Залежно від області дії змінна може бути ***локальною*** або ***глобальною***.

Локальна змінна визначена всередині блока (нагадаємо, що блок розташований між фігурними дужками). Область її дії обмежена початком опису змінної та кінцем блока, включаючи усі вкладені блоки. Змінна**,** визначена поза будь-яким блоком, називається ***глобальною***, її областю дії вважається файл, у якому вона визначена від початку опису до його кінця.

Під час опису змінної перед типом можна вказати **клас пам'яті**, що визначає час існування та область видимості програмного об'єкта, тобто змінної. Якщо клас пам'яті не зазначений явно, то він визначається компілятором, виходячи з контексту оголошення.

Час існування змінної може бути постійним (протягом виконання програми) та тимчасовим (протягом виконання блока).

Областю видимості ідентифікатора називається частина тексту програми, з якої можна здійснити звичайний доступ до зв'язаної з ідентифікатором області пам'яті. Найчастіше область видимості збігається з областю дії. Винятком є ситуація, коли у вкладеному блоці

описана змінна з таким же ім'ям. У цьому випадку зовнішня змінна у вкладеному блоці невидима, хоча він і входить до її області дії. Проте до цієї змінної, якщо вона глобальна, можна звернутися, застосовуючи операцію доступу до області видимості — «**::**».

Клас пам'яті задають такі специфікатори:

- **auto** — в мові С автоматична змінна, для якої пам'ять виділяється у стеку та за необхідності ініціюється щоразу під час виконання оператора, що містить її визначення. Звільнення пам'яті відбувається при виході з блока, де описана змінна. Час її існування — з моменту опису до кінця виконання блока. Для глобальних змінних цей специфікатор не використовується, а для локальних він приймається за замовчуванням, тому задавати його явно великого сенсу немає. В C++ 11 **auto** використовується для автоматичного виведення типу даних компілятором. Це дозволяє програмісту уникнути явного вказання типу змінної, що робить код більш читаним та гнучким;
- **extern** означає, що змінна визначена в іншому місці програми (в іншому файлі або далі по тексту) та використовується для створення змінних, доступних в усіх модулях програми, де вони оголошені. При ініціюванні змінної у тому ж операторі специфікатор **extern** ігнорується;
- **static** — *статична* змінна, що має постійний час існування. Вона ініціюється один раз при першому виконанні оператора, що містить визначення змінної. Залежно від розташування оператора, описані статичні змінні можуть бути глобальними та локальними. Глобальні статичні змінні видимі тільки у тому модулі, в якому вони описані;
- **register** — аналогічний до специфікатора *auto*, але пам'ять виділяється по можливості в регістрах процесора та за відсутності такої можливості у компілятора змінні обробляються як *auto*. Ключове слово *register* було спочатку введено в мові С, щоб рекомендувати компілятору використовувати для зберігання автоматичної змінної регістр центрального процесора. Ідея полягала в тому, що це прискорювало доступ до змінної. До появи стандарту C++ 11 це ключове слово застосовувалося в C++ схожим чином, але з однією відмінністю. Оскільки обладнання та компілятори стали більш досконалими, ця рекомендація була узагальнена і почала вказувати на той факт, що змінна інтенсивно використовується, і, можливо, компілятор зуміє приділити їй особливу увагу. В C++ 11 ця рекомендація є застарілою та ключове слово *register* залишається просто способом ідентифікувати змінну як автоматичну. З огляду на те, що *register* може застосовуватися тільки зі змінними, які будуть автоматичними в будь-якому випадку, одна з причин використання цього ключового слова — вказати, що дійсно потрібна

автоматична змінна, можливо, з тим же самим ім'ям, що і у зовнішньої змінної. Точно таким же було первісне призначення *auto*. Однак більш важлива причина того, що ключове слово *register* залишилося, пов'язана з бажанням зберегти допустимим існуючий код, в якому воно використовується.

Наведемо фрагмент програми з використанням розглянутих вище понять:

```
    int d;        // глобальна змінна d
    extern int y; // змінна y визначена в іншому місці

   int main()
    {
       int b;     // локальна змінна b
       d = 1;     // присвоюємо значення глобальній змінній
       static int s; // локальна статична змінна s
       int d;        // локальна змінна d
       int d = 10;   // присвоюємо значення локальній змінній
       ::d = 3;      // змінюємо значення глобальної змінної
return 0;
     }
   int y = 4;  // визначення та ініціалізація зовнішньої змінної y
```

У цьому прикладі глобальна змінна *d* визначена поза всіма блоками. Пам’ять для неї виділяється в сегменті даних на початку роботи програми, областю дії є вся програма. Область видимості — вся програма. В програмі визначається локальна змінна з тим же ім’ям *d*, область дії якої починається з початку її опису та закінчується при виході з блока. Змінні *b* і *s* — локальні, область їхньої видимості — блок, але час існування різний: пам’ять під *b* виділяється в стеку при вході у блок і звільняється при виході з нього, а змінна *s* розташована у сегменті даних та існує увесь час роботи програми. Якщо початкове значення змінних явно не задається, компілятор присвоює глобальним і статичним змінним нульове значення відповідного типу. Автоматичні змінні не ініціюються. ***Початкове ініціювання змінних не є обов’язковим, проте все ж його бажано здійснювати***.

Опис змінної може виконуватися у формі оголошення або визначення. **Оголошення** інформує компілятор про тип змінної і клас пам’яті, а **визначення** містить, крім цього, вказівку компілятору про виділення пам’яті відповідно до типу змінної. В C++ більшість оголошень є одночасно і визначеннями (у наведеному вище програмному фрагменті тільки опис *extern int y;* є оголошенням, але не визначенням). ***Оголошення тільки описує властивості змінної, а визначення зв’язує її з конкретною областю пам’яті***.

Розглянемо далі основні типи змінних.

Цілі змінні (типу **int**, **long**, **short**) необхідні для збереження цілих значень і можуть бути знаковими та беззнаковими. Знакові змінні застосовують для подання як додатних, так і від'ємних чисел, при цьому один біт (найстарший) виділяється під знак. Для оголошення беззнакової змінної, тобто змінної, що приймає тільки додатні значення, необхідно використовувати ключове слово **unsigned**. За замовчуванням будь-який цілий тип вважається знаковим, і тому немає потреби у використанні ключового слова **signed.**

Символьний тип даних — тип даних, призначений для зберігання одного символу (керуючого або друкованого) у певному кодуванні. Може бути як однобайтовим (для стандартної таблиці символів ASCII), і багатобайтовим (наприклад, для Юнікоду). Основним застосуванням є звернення до окремих знаків рядка.

Змінна типу **bool** займає 1 байт і використовується, насамперед, у логічних операціях, тому що може приймати значення **0** (**false** — «неправда») або відмінне від нуля (**true** — «істина»).

Стандарт C++ визначає три типи даних для збереження ***дійсних значень змінних***: **float, double** та **long double** (типи ***з плаваючою крапкою***). Тип **float**, як правило, використовують для збереження не дуже великих дробових чисел.

Змінна типу **void** не має значення, оскільки множина значень цього типу порожня. Такі змінні необхідні для узгодження синтаксису. Тип *void* використовується для визначення функцій, що не повертають значення, для вказівки порожнього списку аргументів функції, а також як базовий тип для покажчиків і в операції приведення типів. Наприклад, якщо немає потреби у використанні поверненого значення функції, перед ім'ям функції ставиться тип *void*:

```
void minmax(int*x, int k, int*min, int&max);
```

3.3 Константи

Константи є фіксованими значеннями, що не можуть змінюватися впродовж виконання всієї програми.

Спосіб визначення кожної константи залежить від її типу. Константи мови C++ необхідно поділяти на літеральні та типізовані.

Літеральна константа — це лексема, що є зображенням фіксованого числового, рядкового або символьного значення. Такі константи бувають цілі, дійсні, символьні та рядкові (табл. 3.2).

Таблиця 3.2 – Літеральні константи мови С++

Константа	Формат	Приклади
Ціла	Десятковий: послідовність десяткових цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), що починається не з нуля, якщо це не число нуль Вісімковий: нуль, за яким розташовані вісімкові цифри (0, 1, 2, 3, 4, 5, 6, 7) Шістнадцятковий: 0x чи 0X, за яким розташовані шістнадцяткові цифри (0,1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)	3, 0, 21457 03, 051, 0715 0x2B8, 0X0AFF
Дійсна	Десятковий: [цифри] .[цифри] Експоненціальний: [цифри][.][цифри]{E¦e}[+¦-][цифри]	3.1, 03, 81. 0.3E4, 5e-3 6.6, -0.9e-2, 92.E-1, 92E1
Символьна	Один чи два символи, що укладені в апострофи	‘A’, ‘ю’, ‘*’, ‘db’, ‘\0’, ‘\n’
Рядкова	Послідовність символів, що укладена в лапки	"RESULT ", "\t sum=\0x5\n"

Цілі константи можуть бути *десятковими*, *вісімковими* та *шістнадцятковими*.

Довгі цілі константи (**long**) мають літеру **l** або **L** в кінці, наприклад: 32768L; 0777777l; 0XFL. Для завдання константи без знаку (**unsigned**) застосовується літера **u** (**U**), наприклад, 65535u. Довгі константи без знаку записуються з використанням двох літер відразу: (**ul**, **UL**) або (**lu**, **LU**).

Дійсні числа у мовах програмування мають дві форми подання: десяткову (природну) та експоненціальну (показникову).

Десяткова форма дійсного числа — це звичайний десятковий формат запису дійсного числа, тільки частина дійсного числа відділяється від дробової крапкою, а не комою, наприклад: 10.123, 1.0123, 1012.3, 0.0010123.

Експоненціальна форма дійсного числа використовується для запису дуже великих або дуже малих чисел, для яких задавати зайві нулі не зовсім зручно, наприклад: $1.0123 \cdot 10^{20}$, $1.0123 \cdot 10^{10}$. У цій формі запису числа можна виділити такі основні характеристики: знак числа, мантису числа, знак порядку та порядок числа. Зазначені характеристики дійсного числа зберігаються у пам’яті комп’ютера.

Число у показниковій формі може бути представлено, наприклад, так: 1.0123E10. Мантиса записується ліворуч від знаку експоненти (**E** чи **e**), порядок — праворуч. Символ **E** (**e**) означає основу степеня 10 і компілятор розпізнає цей запис як форму представлення дійсного числа. Символ пробілу всередині числа не допускається, а для відділення цілої частини від дробової використовується не кома, а крапка. При додатних значеннях числа та мантиси знак «**+**» можна не вказувати.

Як десяткова, так і експоненціальна форми запису допускають відсутність або цілої частини, або дробової, але не обох одразу.

За замовчуванням всі дійсні константи мають тип **double** — подвійну точність, що найчастіше займає в пам'яті 64 біти, тобто 8 байтів. Але у випадку, якщо програміста не влаштовує тип за замовчуванням, його можна вказати явно за допомогою спеціальних літер. Так, додавши літеру **f** чи **F**, константі надають дійсний тип **float** зі звичайною точністю, наприклад, 8.5f. Якщо в представленні константи використовується літера **L** чи **l**, то вона має тип **long double**.

Зображення від'ємної цілої чи дійсної константи вважається константним виразом, що складається зі знаку унарної операції зміни знаку (-) та константи, наприклад: -273, -2730.e1, -273L.

Символьні константи мають один або два символи, що подаються в апострофах. Односимвольні константи займають у пам'яті один байт і мають стандартний тип **char** (*character* — символ). Двосимвольні константи займають два байти та мають тип *int*. Символьні константи мають цілий тип і їх можна використовувати як цілочислові операнди у виразах.

Заслуговують на увагу послідовності, що починаються зі знаку «****», їх називають **керуючими** або **escape-послідовностями**. Символ зворотної косої риски «****» використовується для запису кодів, що не мають графічного зображення, для запису символів, а також для виведення символьних констант, якщо їх коди задані у вісімковому та шістнадцятковому вигляді (табл. 3.3).

Рядкова константа (***рядковий літерал***) — це послідовність символів, що подається в лапках (тобто в символах «**»**») і зберігається у неперервній ділянці пам'яті, наприклад: *«Це рядковий літерал»*. У кінець кожного рядкового літералу компілятором додається нуль-символ, що представляється керуючою послідовністю *«\0»*. Тому довжина рядка завжди на одиницю більше кількості символів у його записі. Таким чином, порожній рядок (**«»**) має довжину 1 байт. Необхідно звернути увагу на різницю між рядком з одного символу, наприклад, **«**C**»** і символьною константою **'**C**'**. Порожня символьна константа неприпустима.

Таблиця 3.3 – Керуючі послідовності мови С++

\a	звуковий сигнал
\f	переведення сторінки (формату)
\n	новий рядок
\r	повернення каретки
\t	горизонтальна табуляція
\v	вертикальна табуляція
****	символ «****» — зворотна коса риса
\'	символ «**'**» — апостроф
\»	символ «**»**» — лапки
\0	нуль-символ
\?	Знак питання
\0ddd	вісімковий код символу
\0xddd	шістнадцятковий код символу

Керуючі послідовності можуть також застосовуватися у рядкових константах. Так, якщо всередині рядка потрібно записати лапки, то перед ними треба розташувати зворотну косу риску («****»), за якою компілятор відрізняє їх від лапок, що обмежують рядок:

```
«Книга має назву \«Мова програмування С++\» » .
```

Рядки, що записані у програмі підряд або через символи пробілу, при компіляції конкатенуються (фактично «склеюються»). Тобто послідовність двох рядків

```
«Не кажи – не вмію, а кажи – навчусь!»
«Наука не йде на бука!»
```

цілком еквівалентна рядку:

```
«Не кажи – не вмію, а кажи – навчусь! Наука не йде на бука!»
```

Довгу рядкову константу можна розмістити також на декількох рядках. У цьому випадку ставиться зворотна коса риска та натискається клавіша **Enter**. Наприклад:

```
«Комп'ютерна програма виконує те,\
що ви їй наказали робити, а не те, \
що ви бажали, щоб вона робила.»
```

Поняття та приклади оголошення **типізованої константи**, тобто константи, що використовується як змінна, значення якої не може бути змінене після ініціювання, розглянуті вище.

Існує інша можливість задання констант — з використанням директиви препроцесора **#define**, при цьому оголошення має вигляд:

```
#define ім'я константи значення константи
```

і наприкінці такого запису символ «**;**» не ставиться, тобто:

```
#define max 65532
#define km 1000
```

Директива **#define** визначає ідентифікатор (**ім'я константи**) та послідовність символів (**значення константи**), яка замінює ідентифікатор у тексті програми.

Під час застосування великої кількості логічно взаємозалежних констант C++ доцільно користуватися **константами переліку**.

Тип переліку має вигляд:

```
enum {список іменованих констант};
```

— неіменований перелік;

```
enum [ім'я] {список іменованих констант};
```

— іменований перелік,

де **enum** — службове слово (***enumerate*** — перелічувати);

ім'я — ім'я списку констант;

список іменованих констант — розділена комами послідовність ідентифікаторів або іменованих констант вигляду:

```
ім'я константи = значення константи;
```

Наприклад:

```
enum { off, on };
enum Months { January = 1, February, Marth, April, May, June, July,
August, September, October, November, December };
```

Якщо значення константи переліку не визначено, то воно на одиницю більше значення попередньої константи. За замовчуванням перша константа має значення 0. Тоді у першому прикладі константи одержать значення: *off = 0, on=1*, а у другому — значення: *January = 1, February = 2, Marth = 3* тощо. Іменований перелік задає унікальний цілочисельний тип і може використовуватися як специфікація типу для визначення змінних.

Традиційний перелік у C++ дозволяє визначити набір іменованих цілих констант без обмеження їхнього базового типу. Це означає, що значення не інкапсулюються у певну область видимості, що призводить до потенційних конфліктів просторів імен.

C++11 ввів **клас enum** (також відомий як перелік з областю дії чи сильний перелік) для вирішення проблем традиційного переліку. Клас *enum* забезпечує більш високу безпеку типів, інкапсулюючи значення в їхній власній області дії. Це робить його менш схильним до помилок і більш самодостатнім.

Традиційні переліки не забезпечують суворої безпеки типу. Вони дозволяють неявні перетворення типів між типом *enum* і цілими типами, що може призвести до неочікуваної поведінки.

Класи *enum* пропонують сильну безпеку типів. Вони не допускають неявних типів перетворень, допомагаючи запобігти непередбаченим призначенням або зіставленням між різними *enum*-типами.

```
#include <iostream>
enum class Color { Red, Green, Blue};

int main()
{ Color myColor = Color::Red; //значення enum має область видимості
  int Green = 42;             // жодного конфлікту імен
        std::cout << (int)myColor << std::endl; // друкує 0 (Red)
return 0;}
```

3.4 Представлення змінної/об'єкта в пам'яті

Вся пам'ять розбита по байтах. У кожного байту є своя адреса. При створенні об'єкта виділяється пам'ять відповідно до його типу. Адресі першого байту буде надано ім'я. Значення адреси можна отримати за допомогою оператора "**&**" (амперсанд). Цей оператор повертає значення першого байту виділеної пам'яті під об'єкт.

Розглянемо приклад створення змінної *double x*:

```
#include <iostream>
int main()
{
    setlocale(LC_ALL, "Ukr"); //для української мови
double x= 12.34;
    std::cout<<"Значення змінної/об'єкта = "<<x<<std::endl;
    std::cout<<"Адреса змінної/об'єкта   = "<<&x<<std::endl;
return 0;
}
```

Приклад результату програми:

```
Значення змінної/об'єкта = 12.34
Адреса змінної/об'єкта   = 012FFAD4
```

Це означає, що в пам'яті під об'єкт *x* було виділено 8 байт пам'яті, адреса першого *0x012FFAD4*.

Хоча байт у комп'ютері — неділима частина, він логічно поділяється на біти. **Біт** — одиниця виміру кількості інформації. Може приймати тільки два взаємовиключних значення: «так» або «ні», «1» або «0», «включено» або «вимкнено». Байт у звичайному

комп'ютері поділяється на 8 біт. Це зумовлено тим, що одні з перших процесорів мали шину з восьми проводів. По кожному проводу можна передати два сигнали (у двійковому процесорі) 1 — є напруга, 0 — немає напруги. Відповідно по 8-ми проводах можна закодувати 256 варіантів від 0 до 255:

bin	dec	hex	oct
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	8	10
1001	9	9	11
1010	10	A	12
1011	11	B	13
1100	12	C	14
1101	13	D	15
1110	14	E	16
1111	15	F	17

де bin — означає бінарну (двійкову) систему,

dec — десяткову систему,

hex — шістнадцяткову систему,

oct — вісімкову систему.

Таким чином, шістнадцяткова система — це компактна форма запису двійкової системи.

3.5 Операції

Для здійснення маніпуляцій з даними мова **C++** застосовує широкий набір операцій (табл. 3.4), що виконують формування та, відповідно, подальше обчислення виразів. Вирази містять одну або декілька операцій, об'єкти яких називають операндами. **Операції** становлять деяку дію, що виконується над одним (унарні) або декількома (бінарні, тернарні) операндами, і мають позначення (наприклад, операція перевірки на рівність має позначення «**==**», операція обчислення залишку від ділення цілих чисел має позначення «**%**» тощо).

Операції поділяються на:

- ***унарні*** або ***одномісні*** — **&**, *****, **-**, **+**, **~**, **!**, **++**, **--**, **sizeof**;
- ***бінарні*** або ***двомісні*** — **+**, **-**, *****, **/**, **%**, , **>>**, **&**, **:**, **^**, <, **>**, <=, ==, **>**=, **!**=, **&&**, **||**, =, *****=, **/**=, **%**=, **+**=, **-**=, <<=, **>>**=, **&**=,**|**=, **^**=, **.**, **->**, **,**, **( )**, **[ ]**;
- *умовну* ***тернарну (тримісну)*** операцію **?:** .

Таблиця 3.4 – Основні операції мови С++

№	Операції	Порядок виконання
1	**( ) , { } -> .**	Л → П
2	**! ~ ++ -- & * (type)**	П → Л
3	**sizeof**	П → Л
4	*** / %**	Л → П
5	**+ -**	Л → П
6	**<< >>**	Л → П
7	**< <= > >=**	Л → П
8	**= = ! =**	Л → П
9	**&**	Л → П
10	**^**	Л → П
11	**|**	Л → П
12	**&&**	Л → П
13	**||**	Л → П
14	**?:**	П → Л
15	**= += *= - = /= %=**	П → Л
16	**,**	Л → П

Порядок застосування операції визначається пріоритетом операції (яка операція виконується раніше, а яка пізніше) та асоціативністю (виконується зліва направо або справа наліво). Насамперед реалізуються операції з найвищим пріоритетом.

У таблиці 3.4 літерою «**Л**» позначено операнд, що стоїть ліворуч від знаку операції, літерою «**П**» — операнд, який розташований праворуч від знаку операції, а символом «→» — напрямок виконання операції. Розглянемо основні операції.

Результатом виконання операції є значення.

Можна сказати, що операція, яка використовується найчастіше, — це операція присвоювання. Операції присвоювання дозволяють присвоїти деяке значення. Базова операція присвоювання «**=**» дозволяє ***присвоїти значення правого операнду лівому операнду***:

```
int x;
x = 2;
```

Необхідно зазначити, що тип значення правого операнду не завжди може співпадати з типом лівого операнду. У цьому випадку компілятор намагається перетворити значення правого операнду до типу лівого операнду.

При цьому операції присвоювання мають **правобічний порядок**, тобто виконуються справа наліво. І, таким чином, можна виконувати множинне присвоювання:

```
int a, b, c;
a = b = c = 123;
```

Тут спочатку обчислюється значення виразу *c = 123*. Значення правого операнду 123 присвоюється лівому операнду c. Далі обчислюється вираз *b = c*: значення правого операнду c (123) присвоюється лівому операнду b. І в кінці обчислюється вираз *a = b*: значення правого операнду *b* (123) присвоюється лівому операнду *a*.

Крім того, необхідно зазначити, що операції присвоювання мають найменший пріоритет порівняно з іншими типами операцій, тому виконуються в останню чергу:

```
int x;
x = 13 + 25;
```

Згідно з пріоритетом операцій спочатку виконується вираз *13+25*, і тільки потім його значення присвоюється змінній *x*.

Загальний вигляд операції присвоювання при ініціалізації змінних:

```
тип ім‘я_змінної = вираз_або_значення;
```

Наприклад:

```
// Ініціалізація змінних
     int a = 18, b = 225;
     double c = 3.550093;
     bool f = false;
     char sym = 'B';
// Ініціалізація змінних з виразом
     float x = -3.5;
     float y = x + 12.8;
     float z = x*x + y*y*y;
     float zz = z + 5.5;
```

Арифметичні операції:

У мові C/C++ підтримуються такі бінарні арифметичні операції:

+ – додавання;

– – віднімання;

***** – множення;

/ – ділення;

% – остача від ділення.

Операції додавання (**+**) та віднімання (**–**) можуть бути як бінарними, так і унарними. Бінарні операції **+** та **–** використовуються у виразах під час проведення обчислень. Унарні операції **+** та **–** використовуються для позначення знаку числа (додатне число або від‘ємне число). Наприклад:

```
int a, b;
a = -8;       //унарна операція «-», позначає знак числа
b = +5;       //унарна операція «+», позначає знак числа, b = 5;
a = b – 3;    //бінарна операція «-», використовується для обчислення
              //виразу
```

Операція ділення «**/**» має свої особливості, які полягають в наступному: якщо два операнди мають цілочисельний тип, то результат буде цілого типу. У цьому випадку відбувається ділення націло, остача від ділення відкидається.

Наприклад:

```
#include <iostream>
int main()
{   int a = 7, b = 2;
    std::cout<<"a = 7, b = 2; a/b = "<< a/b <<std::endl;
    std::cout<<"a = 7, b = 2; b/a = "<< b/a <<std::endl;
return 0;
}
```

Результат:

```
a = 7, b = 2; a/b = 3
a = 7, b = 2; b/a = 0
```

Якщо ж один з операндів має тип з плаваючою комою, тоді результат має також тип з плаваючою комою:

```
#include <iostream>
int main()
{   double a = 7, b = 2;
    std::cout<<"a = 7, b = 2; a/b = "<< a/b <<std::endl;
    std::cout<<"a = 7, b = 2; b/a = "<< b/a <<std::endl;
return 0;   }
```

Результат:

```
a = 7, b = 2; a/b = 3.5
a = 7, b = 2; b/a = 0.285714
```

Порівняйте те ж саме без оголошення змінних:

```
#include <iostream>
int main()
{
    std::cout << "7/2 = " << 7/2 << std::endl;
    std::cout << "2/7 = " << 2/7 << std::endl;
return 0;
}
```

Результат:

```
7/2 = 3
2/3 = 0
```

І з дійсними числами:

```
#include <iostream>
int main()
{
    std::cout << "7./2 = " << 7./2 << std::endl;
    std::cout << "2/7. = " << 2/7. << std::endl;
return 0;
}
```

Результат:

```
7./2 = 3.5
2/7. = 0.285714
```

Операція взяття остачі від ділення «**%**» використовується над цілими операндами. Операція **%** дозволяє отримати остачу від ділення цілих операндів, завжди є цілим числом:

```
#include <iostream>
int main()
{
    int a, b, c;
    a = 15, b = 6;
    c = a % b;
    std::cout << "15 % 6 = " << c << std::endl;
    c = b % a;
    std::cout << "6 % 15 = " << c << std::endl;
return 0;
}
```

Результат:

```
15 % 6 = 3
6 % 15 =  6
```

Цікавою є реалізація цієї операції з операндами різних знаків:

```
#include <iostream>
int main()
{
    std::cout << "-7 % 4 = " << -7 % 4 << std::endl;
    std::cout << "7 % -4 = " << 7 % -4 << std::endl;
    std::cout << "-7 % -4 = " << -7 % -4 << std::endl;
return 0;
}
```

Результат:

```
-7 % 4 = -3
7 % -4 = 3
-7 % -4 = -3
```

Бінарні арифметичні операції можуть бути об'єднані з операцією присвоювання:

```
об'єкт *= вираз;    // об'єкт = об'єкт * вираз
об'єкт /= вираз;    // об'єкт = об'єкт / вираз
об'єкт += вираз;    // об'єкт = об'єкт+ вираз
об'єкт -= вираз;    // об'єкт = об'єкт - вираз
об'єкт %= вираз;    // об'єкт = об'єкт % вираз
       s += 7;              // s =s + 7;
       i *= j + 5;          // i =i*(j + 5);
       g %= 9;              // g =g % 9;
```

Ще приклад:

```
int a = 5;
a += 10;            // 15
a -= 3;             // 12
a *= 2;             // 24
a /= 6;             // 4
a % = 7;            // 4
a % = 3;            // 1
```

Оператори інкременту (++), декременту (––) здійснюють збільшення або зменшення цілочисельної величини на 1:

++ — інкремент;

–– — декремент.

Ці оператори є унарними. Вони вимагають одного операнда. Ці оператори можуть розміщуватися **до** та **після** операнда.

При використанні операції «**++**» перед ім'ям змінної (**преінкремент**) значення змінної спочатку збільшується на 1, а потім використовується у виразі.

При використанні операції «**++**» після імені змінної (**постінкремент**) значення змінної спочатку використовується у виразі, а потім збільшується на 1.

При використанні операції **предекремента** («**--**» перед іменем змінної), значення змінної спочатку зменшується на 1, а потім використовується у виразі.

При використанні операції **постдекремента** («**--**» після імені змінної), значення змінної спочатку використовується у виразі, а потім зменшується на 1.

Приклади використання з результатами:

```
int a, b;
a = 10;
b = a++;        // b = 10; a = 11
a = 10;
b = ++a;        // b = 11; a = 11
a = 10;
```

```
b = a--;        // b = 10; a = 9
a = 10;
b = --a;        // b = 9; a = 9
```

Ще приклад:

```
#include<iostream>
int main()
{
        int a = 35;
        int b = 10;
        std::cout << "a + b++ =" << a + b++ << "\n";
        std::cout<<"b = "<< b << "\n";
        std::cout << "a + --b =" << a + --b << "\n";
        std::cout << "b = " << b << "\n";
return 0;
}
```

Результат виконання:

```
a + b++ =45
b = 11
a + --b =45
b = 10
```

Операції відношення (порівняння) порівнюють значення **Л** зі значенням **П**, використовуються для порівняння двох величин між собою, якими можуть бути числа, змінні, константи, результати обчислення виразу, тощо:

< — менше;

<= — менше або дорівнює (не перевищує);

== — дорівнює;

> — більше;

>= — більше або дорівнює (не менше);

!= — не дорівнює.

У мові С++ **«істина»** — це ненульова величина, **«неправда»** — це нуль (**0**). У більшості випадків одиниця (**1**) використовується як ненульове значення.

Операції відношення повертають ціле значення 1, якщо умова правильна, або 0, якщо умова помилкова.

```
#include <iostream>
int main()
{
    int x = 2, y, z;
    y = x > 1;
    z = x == y;
    std::cout << "x = 2, x > 1 = " << y  << std::endl;
    std::cout << "y = "<<y<< ", x == y = " << z << std::endl;
return 0;
}
```

Результат:

```
x = 2, x > 1 = 1
y = 1, x == y = 0
```

Логічні операції оперують з цілими розмірами або з розмірами, які можна перетворити на цілі. Обчислення зупиняється, як тільки визначиться, чи є вираз правдивим («істина») або помилковим («неправда»). При цьому, як і для операцій відношення, значенням «істина» відповідає **1**, а значенням «неправда» — **0**.

&&— логічне «**AND**» (кон'юнкція);

|| — логічне «**OR**» (диз'юнкція);

!= — логічне «**NOT**» (заперечення).

Результат операції «**&&**» є «істина» (**1**), якщо обидва її операнди правдиві (не рівні 0). Результат операції «**||**» — «істина» (**1**), якщо хоча б один з її операндів є «істина». Логічне заперечення «**!**=» перетворює свій операнд на «істину» (**1**), якщо він дорівнює **0**, і на «неправду» (**0**), якщо він не дорівнює **0**.

Таблиця істинності логічних операцій **&&** (логічне «І»), **||** (логічне «АБО»), **!** (логічне «НІ») має такий вигляд:

Таблиця 3.5 – Таблиця істинності логічних операцій

Операнд a	Операнд b	a&&b	a||b	!a	!b
true	true	true	true	false	false
true	false	false	true	false	true
false	true	false	true	true	false
false	false	false	false	true	true

```
// логічні операції
bool c;
int a, b;
// операція && (AND)
a = 8; b = 5;
c = a && b;     // c = true, тому що a та b не дорівнюють 0
a = 0;
c = a && b;     // c = false
// операція || (OR)
a = 0; b = 0; c = a || b;       // c = false
b = 7; c = a || b;              // c = true
// операція ! (логічне "НІ")
a = 0; c = !a;          // c = true
a = 15; c = !a;         // c = false
```

З використанням логічних операцій та операцій відношення записуються різні умовні вирази, наприклад, умова *3 < x < 5* матиме вигляд: *x > 3 && x < 5*.

Операції обробки окремих бітів застосовують для обробки даних як послідовностей бітів (розрядів), кожний з яких набуває значення **0** або **1**.

& — операція бітового множення (кон'юнкція);

| — операція бітового додавання (диз'юнкція);

^ — додавання за модулем 2;

~ — інвертування;

>> — зсув праворуч;

<< — зсув ліворуч.

Операції **&**, **^**, **|** є бінарними. Це означає, що вони потребують двох операндів. Біти кожного операнда порівнюються між собою за таким правилом: біт в позиції 0 першого операнда порівнюється з бітом у позиції 0 другого операнда. Потім біт у позиції 1 першого операнда порівнюється з бітом у позиції 1 другого операнда. Так порівнюються усі біти цілочисельних операндів.

Кожен біт результату визначається на основі двох операндів, які є бітами, так як показано в таблиці 3.6:

Таблиця 3.6 – Операції обробки окремих бітів

біт 1	**біт 2**	**&**	**||**	**^**	**~ біт1**	**~ біт 2**
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Мови C/C++ включають дві операції порозрядного зсуву:

<< – зсув вліво значення операнду на задану кількість біт. Операнд розміщується зліва від знаку операції (<<). Кількість біт, на які зсувається значення, задаються справа від знаку операції;

>> – зсув вправо значення операнду на задану кількість біт. Операнд розміщується зліва від знаку операції (<<). Кількість біт, на які зсувається значення, задаються справа від знаку операції;

Висувні біти втрачаються, а "всуваються" нульові біти. Зсув операндів вліво на 1, 2, 3 і більше розрядів – найбільш швидкий спосіб множення на 2, 4, 8, … Зсув операндів вправо на 1, 2, 3 і більше розрядів – найбільш швидкий спосіб ділення на 2, 4, 8, …

Якщо у програмі потрібно, щоб операція множення цілочисельних операндів на 2, 4, 8 і т. д. відбувалась максимально швидко, то доцільно використовувати операцію зсуву вліво.

Те саме стосується, якщо потрібно максимально швидко поділити цілочисельний операнд на 2, 4, 8 і т.д. Тоді рекомендовано використовувати зсув вправо.

Змінна-покажчик зберігає значення, що є адресою об'єкта в пам'яті комп'ютера. Через покажчик можна звертатися до об'єкта.

Операції з адресами та покажчиками:

& — одержання адреси: видає адресу змінної, ім'я якої розташоване праворуч від позначення операції;

***** — непряма адресація (розіменування): видає значення, записане за адресою, на яку посилається покажчик.

Додаткові операції:

sizeof() — знаходить розмір (у байтах) операнда, розташованого праворуч від назви операції, наприклад:

```
int x = sizeof(x); //x = 4
```

(type) — операція приведення типу перетворює наступне за нею значення в тип, визначений ключовим словом, укладеним у круглі дужки, наприклад:

```
int i=6;
double Pi = 3.14;
…
i = i+(int)Pi;
```

?: — **тернарна** (з трьома операндами) операція, що має вигляд:

```
вираз1 ? вираз2 : вираз3;
```

Тут, якщо результат обчислення першого операнда (**вираз 1**) не дорівнює **0** («істина»), то результатом операції буде значення другого операнда (**вираз 2**), інакше — третього операнда (**вираз 3**). Наприклад, знаходження найбільшої з двох величин **a** і **b**, можна здійснити операцією:

```
max = (b > a)? b : a;
```

Тут змінна *max* прийме значення *a* або *b* залежно від умови.

Мова C++ налічує широкий спектр математичних функцій (табл. 3.7). Для їхнього використання необхідно включити в код програми заголовний файл **cmath**.

Таблиця 3.7 – Математичні функції (заголовний файл cmath)

Прототип функції	**Ім'я**	**Призначення**
double sin (double _x);	**sin (x)**	синус x (в радіанах)— **sin(x)**
double cos (double _x);	**cos (x)**	косинус x (в радіанах)— **cos(x)**
double tan (double _x);	**tan (x)**	тангенс x(в радіанах)— **tg(x)**
double asin (double _x);	**asin (x)**	арксинус x **— arcsin(x)**
double acos (double _x);	**acos (x)**	арккосинус x — **arccos(x)**
double atan (double _x);	**atan (x)**	арктангенс x — **arctg(x)**
double atan2 (double y, double_x);	**atan2 (y,x)**	арктангенс y/x **— arctg (y/x)**
double sinh (double _x);	**sinh (x)**	синус гіперболічний x — **sh(x)**
double cosh (double _x);	**cosh (x)**	косинус гіперболічний x — **ch(x)**
double tanh (double _x);	**tanh (x)**	тангенс гіперболічний x — **th(x)**
double log (double _x);	**log (x)**	натуральний логарифм x — **ln(x)**
double log10 (double _x);	**log10 (x)**	десятковий логарифм x — **log(x)**
double exp (double _x);	**exp (x)**	піднесення e до степеня x — $\mathbf{e^x}$
double pow (double _x, double_y);	**pow (x, y)**	піднесення x до степеня y — $\mathbf{x^y}$
double pow10 (int _p)	**pow10 (p)**	повертає $\mathbf{10^p}$
double sqrt (double _x);	**sqrt (x)**	корінь із x, **x>0**—$\sqrt{\mathbf{x}}$
double hypot (double_x, double_y);	**hypot(x, y)**	корінь із(x^2+y^2) — $\sqrt{\mathbf{x^2+y^2}}$
double fabs (double _x);	**fabs (x)**	абсолютне значення x –**|x| типу double**
int abs (int _x);	**abs (x)**	абсолютне значення x –**|x| типу int**
long labs (long _x);	**labs (x)**	абсолютне значення x –**|x| типу long**
double fmod (double _x, double_y);	**fmod (x, y)**	повертає залишок ділення **x** на **y** дійсного типу
double ceil (double _x);	**ceil (x)**	округлення до більшого
double floor (double _x);	**floor (x)**	повертає найближче ціле, не більше за **x**
double modf (double _x, double);	**modf(x,&p)**	виділяє цілу й дробову частини числа
double atof(const char*_s);	**atof (s)**	перетворює рядок символів у число з плаваючою крапкою
визначені константи: **M_PI = 3.1415… — Π, M_E = 2.71828… — e, M_SQRT2 = 1.4142… —** $\sqrt{2}$**, M_LN2 = 0.6931… —** $\ln(2)$ тощо		

3.6 Запитання та завдання

1. Які основні та складені типи даних має мова С++?
2. Що таке змінна і як здійснюється її опис та визначення?
3. Що таке «область дії ідентифікатора» та «клас пам'яті»?
4. Які константи налічує С++? Охарактеризуйте їхнє застосування.
5. Що таке пріоритет операції? Наведіть приклади арифметичних та логічних операцій.
6. Які операції присвоювання та операції відношення налічує С++?
7. Що реалізують логічні операції та операції обробки окремих бітів?
8. Які операції над покажчиками та додаткові операції має С++?

Завдання.

Набуття практичних навичок роботи зі змінними, основними операціями та математичними функціями мови С++ при програмній реалізації завдань розділу вправ «Обчислення математичних виразів».

4 ОРГАНІЗАЦІЯ ВВЕДЕННЯ-ВИВЕДЕННЯ ДАНИХ

У мові С++ дії, що пов'язані з операціями введення та виведення, виконуються за допомогою функцій бібліотек. Функції введення та виведення бібліотек мови дозволяють читати дані з файлів та пристроїв і писати дані у файли та на пристрої.

Бібліотека мови С++ підтримує три рівні введення-виведення даних:

- введення-виведення потоку;
- введення-виведення нижнього рівня;
- введення-виведення для консолі та порту.

При введенні-виведенні потоку всі дані розглядаються як потік окремих байтів. Для користувача потік — це файл на диску або фізичний пристрій, наприклад, дисплей чи клавіатура, або пристрій для друку, з якого чи на який направляється потік даних. Операції введення-виведення для потоку дозволяють обробляти дані різних розмірів і форматів від одиночного символу до великих структур даних. Програміст може використовувати функції бібліотеки, розробляти власні та включати їх у бібліотеку. Для доступу до бібліотеки цих класів треба включити в програму відповідні заголовні файли.

За замовчуванням стандартні введення та виведення повідомлень про помилки належать до консолі користувача (клавіатури та екрану). Це означає, що завжди, коли програма очікує введення зі стандартного потоку, дані повинні надходити з клавіатури, а якщо програма виводить дані — то на екран.

4.1 Потокове введення-виведення

У мові С++ існує декілька бібліотек, які містять засоби введення - виведення, наприклад: **stdio.h, iostream**. Найчастіше застосовують потокове введення-виведення даних, операції якого включені до складу класів **istream** або **iostream**. Доступ до бібліотеки цих класів здійснюється за допомогою використання у програмі директиви компілятора **#include <iostream>**.

Для потокового введення даних вказується операція «**>>**» («читати з»). Це перевантажена операція, визначена для всіх простих типів і покажчика на **char**. Стандартним потоком введення є **cin**.

Формат запису операції введення має вигляд:

```
cin >> values;
```

де **values** — змінна.

Так, для введення значень змінних *x* і *y* можна записати:

```
cin >> x >> y;
```

Кожна операція «**>>**» передбачає введення одного значення. При такому введенні даних необхідно дотримуватись конкретних вимог:

– для послідовного введення декількох чисел їх необхідно розділяти символом пробілу (« ») або **Enter** (дані типу **char** розділяти пробілом необов'язково);

– якщо послідовно вводиться символ і число (або навпаки), пробіл треба записувати тільки в тому випадку, коли символ (типу **char**) є цифрою;

– потік введення ігнорує пробіли;

– для введення великої кількості даних одним оператором їх можна розташовувати в декількох рядках (використовуючи **Enter**);

– операція введення з потоку припиняє свою роботу тоді, коли всі включені до нього змінні одержують значення. Наприклад, для операції введення *x* і *y*, що вказана вище, можна ввести значення *x* та *y* таким чином:

```
2.345      789
```

або

```
2.345 «Enter»
789
```

Оскільки в цьому прикладі пробіл є роздільником між значеннями, що вводяться, то при введенні рядків, котрі містять пробіли у своєму складі, цей оператор не використовується. У такому випадку треба застосовувати функції **getline()**, **get()** тощо (розділ 7). У мові C++ бажано здійснювати потокове введення-виведення даних.

Для потокового виведення даних необхідна операція «**<<**» («записати в»), що використовується разом з ім'ям вихідного потоку **cout**. Наприклад, вираз

```
cout << x;
```

означає виведення значення змінної *x* (або запис у потік). Ця операція вибирає необхідну функцію перетворення даних у потік байтів.

Формат запису операції виведення представляється як:

```
cout << data [<< data1];
```

де *data, data1* — це змінні, константи, вирази тощо.

Потокова операція виведення може мати вигляд:

```
cout << "y =" << x + a - sin(x) << "\n";
```

Застосовуючи логічні операції, вирази треба брати в дужки:

```
cout << "p =" << (a && b || c) << "\n";
```

Символ переведення на наступний рядок записується як рядкова константа, тобто *"\n"*, інакше він розглядається не як символ керуючої послідовності, а як число 10 (код символу).

Треба пам'ятати, що ***при виведенні даних з використанням «cout <<» не виконується автоматичний перехід на наступний рядок, для реалізації такого переходу застосовується знак переведення рядка* "\n" *або операція* endl**. Тобто, вивести рядкову константу можна, наприклад, так:

```
cout << " It's never too late to learn \n";
```

або

```
cout << " It's never too late to learn "<< endl;
```

Приклад 4.1.

Написати програму, що містить виведення даних, пояснювальні повідомлення, а також символи переведення рядка.

```
#include <iostream>
int main()
{
char symbol1 = 'W';
char symbol2 = 'P';
char symbol3 = 'S';
int line1 = 20;
int line2 = 2;
float weight1 = 309.75;
float weight2 = 8.5;
        // виведення результатів
std::cout <<"Data verification\n";
std::cout<< symbol1 << symbol2 << symbol3 << "\n\n";
std::cout<<"LINE1  LINE2  WEIGHT1 WEIGHT2:\n";
std::cout <<" " <<line1;
std::cout <<"      "<<line2<<"     "<<weight1<<"  "<<weight2;
return 0;
}
```

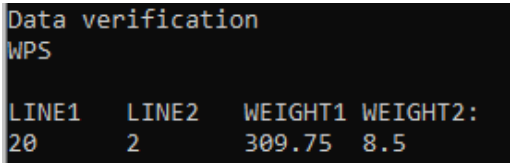
Результат:

```
Data verification
WPS

LINE1  LINE2  WEIGHT1 WEIGHT2:
 20       2     309.75  8.5
```

В останніх двох операціях виведення програми можна використати символи табуляції. Наприклад, **«\t»** поміщає кожне наступне ім'я або число в наступну позицію табуляції (через вісім символів), у цьому випадку маємо:

```
std::cout<<"LINE1\tLINE2\tWEIGHT1\tWEIGHT2:\n";
std::cout<<line1<<"\t"<<line2<<"\t"<<weight1 <<"\t"<<weight2;
```

Тоді результат роботи програми буде такий:



Для додаткового керування даними, що виводяться, використовують маніпулятори **setw(w)** та **setprecision(d)**. Маніпулятор **setw(w)** призначений для зазначення довжини поля, що виділяється для виведення даних (**w** — кількість позицій). Маніпулятор **setprecision(d)** визначає кількість позицій у дробовій частині дійсних чисел.

Маніпулятори змінюють вигляд деяких змінних в об'єкті **cout**, що у потоці розташовані за ними. Ці маніпулятори називають *прапорцями стану*. Коли об'єкт посилає дані на екран, він перевіряє прапорці, щоб довідатися, як виконати завдання, наприклад, запис:

```
    cout << 456 << 789 << 123;
```

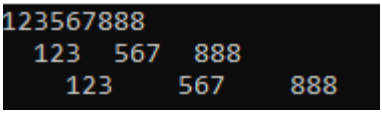
призводить до виведення значення у вигляді: **456789123**, що ускладнює визначення групи значень.

Приклад 4.2.

Написати програму, використовуючи маніпулятор **setw()**.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
cout << 123 << 567 << 888 << endl;
cout << setw(5) <<123 <<setw(5) << 567 << setw(5) << 888 << endl;
cout << setw(7) <<123 <<setw(7) << 567 << setw(7) << 888 << endl;
return 0;
}
```

Результат виконання програми:



У цьому прикладі з'явився новий заголовний файл **iomanip**, що дозволяє застосовувати функції маніпуляторів. При використанні функції **setw()** число вирівнюється вправо в межах заданої ширини поля виведення. Якщо ширина недостатня, то вказане значення ігнорується.

Функція *setprecision(2)* повідомляє про те, що число з плаваючою крапкою виводиться з двома знаками після крапки з округленням дробової частини, наприклад, під час виконання операції

```
cout << setw(7) << setprecision(2) << 123.456789;
```

буде отримано такий результат: **123.46**.

В цій програмі з'являється нова конструкція **`using namespace std;`**, яка дозволяє використовувати стандартний простір імен std (Standard Library) в поточному файлі програми. Це дає можливість використовувати операції та функції без уточнення (порівняйте з `std::cout`).

Функції **cout.width(w)** та **cout.precision(d)**, які потребують підключення тільки заголовного файлу **iostream**, виконують дії, подібні тим, що і функції **setw(w)** та **setprecision(d)**.

Операція введення використовує ті ж самі маніпулятори, що й операція виведення.

Приклад 4.3.

Написати програму обчислення податку на продаж.

```
#include<iostream>
#include<iomanip>
using namespace std;

int main()
{
        float sales_sum;  // сума продажів
        float tax;
        // виведення підказки для користувача
        cout <<"Enter the sales amount  ";
        cin >> sales_sum;
        // обчислення податку на продаж
        tax = sales_sum * 0.7;
```

```
        cout << "\nsales_sum = " << setprecision(2) << sales_sum;
        cout << "\ntax = " << setprecision(2) << tax << "\n";
return 0;
}
```

Унаслідок того, що у першому операторі **cout** відсутня інструкція переведення рядка, відповідь користувача на підказку (тобто введене значення змінної *sales_sum*) з'явиться відразу праворуч за самою підказкою.

Результат:

```
Enter the sales amount  12.345

sales_sum = 12
tax = 8.6
```

Порівняйте результати при заміні останніх двох операцій виведення на наступні (змінили аргумент функції *setprecision(x)*):

```
cout << "\nsales_sum = " << setprecision(5) << sales_sum;
cout << "\ntax = " << setprecision(5) << tax << "\n";
```

Тоді:

```
Enter the sales amount  12.345

sales_sum = 12.345
tax = 8.6415
```

4.2 Форматоване введення-виведення

Форматоване введення-виведення величин здійснюється з використанням функцій **scanf** та **printf**, які успадковані з мови С. Щоб зв'язати програму користувача зі стандартною бібліотекою, де знаходяться ці функція, необхідно на початку програми включити заголовний файл **stdio.h**.

Функція scanf, що забезпечує форматоване введення даних, має змінне число параметрів, при цьому перед відповідним параметром ставиться знак «**&**»— символ взяття адреси змінної. Наприклад, **&x1** означає адресу змінної **x1**, а не значення, яке ця змінна має в даний момент. Рядок форматів функції **scanf** вказує, які дані очікуються на вході. Якщо функція зустрічає у форматному рядку знак «**%**», за яким розташований знак перетворення, то на вході пропускатимуться символи, доки не з'явиться деякий непорожній символ.

Форма запису функції **scanf** має вигляд:

```
scanf ("рядок форматних кодів", список імен змінних);
```

Рядок форматних кодів становить таку структуру запису:

%[*][ширина][довжина]тип,

де «**%**» — позначає початок специфікатора формату;

***** — необов'язковий параметр, якщо він вказаний, то scanf пропустить цей елемент вводу, не записуючи його в жодну змінну;

ширина — максимальна кількість символів, які scanf зчитує для цього конкретного вводу;

довжина — вказує модифікатор довжини, який уточнює розмір типу даних. Найпоширеніші:

h — для коротших типів, наприклад *short int* або *unsigned short int*;

l — для довших типів, наприклад, *long int* або *double*;

ll — для типу *long long int*;

тип — вказує тип даних, які потрібно зчитати. Основні специфікатори:

%d — ціле число (*int*);

%f — число з плаваючою крапкою (*float*);

%lf — число з подвійною точністю (*double*);

%c — символ (*char*);

%s — рядок (масив символів).

Приклад 4.4.

Програма пропускає введення числа з плаваючою крапкою через використання *****.

```
#include <stdio.h>
int main()
{
    int a;
    float b;
    char c;

    printf("Enter an integer, a float, and a character: ");
    scanf("%d %*f %c", &a, &c);
    // Зчитує ціле число, пропускає float, зчитує символ

    printf("Integer: %d, Character: %c\n", a, c);
return 0;
}
```

Результат виконання програми:

```
Enter an integer, a float, and a character: 2 0.234 f
Integer: 2, Character: f
```

Приклад 4.5.

Ввести два числа та обчислити їхню суму.

```
#include <stdio.h>
int main()
{
                int a, b, c;
                scanf("%d %d", &a, &b);     // введення чисел
                c = a + b;
                printf("Sum = %d \n", c);
return 0;
}
```

Результат виконання програми:

```
12 345
Sum = 357
```

Форматний рядок наказує функції *scanf()* ввести десяткове число, яке треба помістити в змінну *a*, а потім перейти до наступного непорожнього символу і з цього моменту почати введення нового десяткового числа, яке потім присвоюється змінній *b*. Якщо за рядком керування форматом аргументів більше, ніж специфікацій формату, зайві аргументи ігноруються. Коли для специфікацій формату недостатньо аргументів, результат не визначений.

У наведеному фрагменті програми для форматованого виведення даних використовується функція **printf**.

Функція printf може використовуватися, наприклад, для виведення повідомлення на екран:

```
printf ("Enter the initial data \n");
```

Для звернення до функції використовуються параметри, які розташовані у круглих дужках. Найчастіше функція **printf** реалізується для виведення значень змінних. Першим аргументом у звертанні до функції ставиться рядок форматів (береться в лапки), а наступними, якщо вони є, — об'єкти, що виводяться.

Рядок форматів може включати звичайні символи, які копіюються при виведенні, і специфікації перетворення, що починаються із символу «**%**», за специфікаціями йде символ перетворення. Кожна специфікація перетворення відповідає одному з аргументів, що йдуть за форматним рядком, і між ними встановлюється взаємно однозначна відповідність, наприклад:

```
printf ("The values of a and c are equal to: %d %d \n", a, c); ,
```

тут літера *d* у специфікації перетворення вказує, що значення аргументу має бути представлено як десяткове ціле число.

При виведенні функція **printf** використовує ті самі специфікації, що й функція **scanf** при введенні. Наприклад, у функції **printf** вигляду

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

значення змінної *g* виводиться як символ алфавіту, а після знаку «=» — як числове значення. Перед символом перетворення може стояти числовий коефіцієнт, що вказує кількість позицій у виведеному рядку, відведених для елемента виведення.

Список форматних кодів має таку форму запису:

```
% [прапорець] [довжина] [.точність] [f | n] [h | l] тип ;
```

де **прапорець** — символ, що керує вирівнюванням виведення та виведенням пробілів, десяткової крапки, ознак чисел вісімкової та шістнадцяткової систем числення. **Прапорець** може задаватися одним із символів:

«**-**» — вирівнювання вліво усередині заданого поля;

«**+**» — виведення знака числа;

« »(пробіл) — приєднання пробілу до виведеного числа, якщо число є додатним і має тип зі знаком;

«**#**» — виводиться ідентифікатор системи числення для цілих: **0** — для вісімкових чисел, **0x** чи **0X** — для шістнадцяткових чисел;

довжина — визначає мінімальну кількість виведених символів, якщо довжина більше виведеної кількості символів, то виведене значення доповнюється пробілами, у випадку, коли довжина менше кількості символів у виведеному значенні або вона не задана, виводяться всі символи значення (відповідно до поля **точність**, якщо воно є);

точність — задається цілим числом після крапки і визначає кількість виведених символів, кількість знаків після крапки; на відміну від поля **довжини** поле **точність** може призвести до «зрізання» виведених даних.

Параметри *f*, *n*, *h*, *l* і *тип* списку форматних кодів за змістом аналогічні раніше описаним для функції *scanf*.

Виведення результатів з використанням форматних кодів функції *printf* може мати вигляд: *printf (" % 3.0 f % 6.1 f \ n ", x, y);* .

Приклад 4.6.

Обчислити значення функції **y** = **ax^2 – sin(x)**, якщо **a** = **10.5**, крок **hx** = **0.5**, проміжок **x** ∈ **[-1; 2]**.

Алгоритм розв'язання поставленої задачі розглянуто у прикладі 1.1, а його програмна реалізація матиме такий вигляд:

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

```
int main()
{       float x, y, a(10.5);
        printf("\t Rezult\n");
        for (x = -1; x <= 2; x += 0.5)
        {       y = a * pow(x, 2) - sin(x);   // y = a*x*x-sin(x);
                printf(" \t x = % 4.1f   y = % 6.3f \n", x, y);
        }
return 0;
}
```

Результати обчислення:

```
 Rezult
x = -1.0   y =  11.341
x = -0.5   y =  3.104
x =  0.0   y =  0.000
x =  0.5   y =  2.146
x =  1.0   y =  9.659
x =  1.5   y =  22.628
x =  2.0   y =  41.091
```

4.3 Використання українських шрифтів у консольних програмах C++ в різних компіляторах

Використання українського шрифту в консольних програмах C++ залежить від компілятора та налаштувань консолі операційної системи. Різні компілятори можуть працювати з різними кодуваннями за замовчуванням, тому для коректного виведення українських символів потрібно враховувати кілька факторів: кодування вихідного файлу, налаштування консолі, та правильну компіляцію.

Visual Studio підтримує роботу з українськими символами через консоль, але необхідно налаштувати кодову сторінку та використовувати функції для роботи з Unicode або кодуванням Windows-1251

Приклад 4.7.

Український шрифт для Visual Studio

```
#include <iostream>
#include <windows.h> //для кодування

int main()
{
// встановлюємо кодову сторінку
// для українських символів Windows-1251
    SetConsoleOutputCP(1251);
    SetConsoleCP(1251);
```

```
/* Якщо вам потрібно використовувати універсальне кодування, таке як
UTF-8, це можна зробити за допомогою функції
SetConsoleOutputCP(CP_UTF8) */

    // Виводимо український текст
    std::cout << "Привіт, світе!" << std::endl;

return 0;
}
```

В інших компіляторах можна спробувати застосовувати функцію **setlocale(LC_CTYPE, "ukr");**, яка використовується для налаштування локалізації вашої програми, зокрема для керування тим, як відбувається обробка символів (типи символів, сортування тощо) на основі вибраної локалі.

4.4 Запитання та завдання

1. Як забезпечується організація введення даних на C++?
2. Як здійснюється виведення результатів на C++?
3. Як реалізується потокове введення даних *cin*>>? Наведіть приклади.
4. Охарактеризуйте основні аспекти використання потокового виведення даних *cout*<<.
5. Що таке форматоване введення-виведення даних?
6. Поясніть правила застосування функції *scanf*.
7. Як здійснює роботу функція *printf*?

Завдання.

Набути практичних навичок при здійсненні введення/виведення даних завдань розділів вправ «Обчислення математичних виразів» та «Циклічні обчислювальні процеси».

5 ОСНОВНІ ОПЕРАТОРИ МОВИ С++

5.1 Умовні оператори

До умовних операторів належать оператор умовного переходу **if**, операторів вибору (перемикач) **switch** та тернарну операцію **?:** [4, 6–8, 12–14, 16].

Оператор умовного переходу *if* використовується для розгалуження процесу обчислень на два напрямки та має такий формат запису:

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

де **вираз** — це вираз, який має логічне значення (**true** — «істина» або **false** — «неправда»).

Реалізується оператор **if** таким чином: спочатку обчислюється **вираз** і, якщо значення виразу не дорівнює нулю («істина»), виконується **оператор 1**, в протилежному випадку — **оператор 2** і далі управління передається оператору, що є наступним за умовним оператором **if**, наприклад:

```
if (i>j) i++;
else
{   j = i - 3;
   i++;  }
```

Наприклад, фрагмент коду, в якому обчислюється значення виразу $y = \begin{cases} x^3 + 2x - 4, \text{ якщо } -8 \le x \le 5 \\ x - 5 \text{ в протилежному випадку} \end{cases}$:

```
float f, x;
// введення x
// ...
if ((-8 <= x) && (x <= 5))
   f = x * x * x + 2 * x - 4;
else
   f = x - 5;
```

Інколи в програмах на С++ повну форму оператора **if** доцільно замінити на скорочену форму. Це необхідно у випадках, коли після слова **else** не потрібно виконувати ніяких інструкцій. В скороченій формі оператора **if** ключове слово **else** опускається.

Загальний вигляд скороченої форми оператора **if**:

```
if (вираз) { /* декілька операторів ...*/};
```

або

```
if (вираз) оператор;
```

де ***вираз*** – умовний вираз (умова) згідно з синтаксисом мови **С++**.

Коли в запису оператора **if** друга частина (тобто **else**) відсутня, якщо вираз приймає значення «неправда», виконується зразу наступний оператор програми, що розташований за умовним. Таку конструкцію називають «пропуск оператора».

Наприклад:

```
#include<iostream>
using namespace std;
int main()
{
        int x = 10, a;
        cout << "Input a = ";
        cin >> a;
        if (a > x) a = x;
        a = -x;
        cout << "\na = " << a;
return 0;
}
```

В наведеній програмі при введені будь-якого значення змінної *a* (більше за *x* чи менше за *x*, чи рівне *x*) буде виведено значення *a = -10*, тому що оператор *a = -x;* буде виконаний в будь-якому випадку:

```
Input a = 1             Input a = 20            Input a = 10

a = -10                 a = -10                 a = -10
```

Коли у будь-якій гілці розгалуження необхідно виконати декілька операторів, їх треба розташовувати у блоці (в фігурних дужках **{}**), інакше компілятор не зможе зрозуміти, де закінчується розгалуження. Блок може включати різні оператори, у тому числі описи та інші умовні оператори, але не може складатися з одних описів. Необхідно ураховувати, що змінна, яка описана у такому блоці, за межами блока не існує. Синтаксис С++ припускає, що у випадку застосування вкладених умовних операторів кожне **else** відповідає найближчому до нього попередньому **if**.

Наприклад:

```
    if (c > d && (c > r || c == 0))
       d++;
    else
    {  d /= c; c = 0;}
```

декілька умов треба об'єднати знаками логічних операцій, а сукупність операцій необхідно розмістити у блоці (додаткових фігурних дужках {}).

Обчислення найбільшого значення серед трьох змінних *a, b, c*:

```
if (a > b)
{   if (a > c) max = a;
    else max = c; }
else
    if (b > c) max = b;
    else max = c; }
```

У цьому фрагменті програми фігурні дужки можуть бути відсутні, тому що компілятор відносить частину **else** до найближчого **if**.

Фігурними дужками можна обмежити дію умовного оператора, зробивши його скороченим.

Порівняйте програми та результати виконання:

```
#include<iostream>
using namespace std;
int main()
{
int a = 1, b = 5, c = 7;
        if (a > b)
            if (b > c) c = b;
            else c = a;
        cout << "c = " << c;
return 0;
}
```

Результати:

```
c = 7
```

```
#include<iostream>
using namespace std;
int main()
{
int a = 1, b = 5, c = 7;
    if (a > b)
     { if (b > c)   c = b;  }
    else c = a;
        cout << "c = " << c;
return 0;
}
```

```
c = 1
```

Оператор-перемикач switch називають оператором множинного розгалуження. Він використовується для вибору одного з багатьох варіантів рішення та має таку форму запису:

```
    switch (L)
    {
       case к.в.1: оператор1; [break; ]
       case к.в.2: оператор2; [break; ]
       . . . . . . . . . . . . . . . . . . . . .
       case к.в.N: операторN; [break; ]
       [default: операторN + 1; ]
    }
```

де **switch, case, default** — службові слова;

break — оператор (необов'язковий) здійснює вихід з оператора **switch**;

L — будь-який вираз одного з цілих типів;

к.в.1, …, к.в.N — константні вирази, які не можуть повторюватися та не можуть містити змінних чи викликів функцій; зазвичай, це ціла або символьна константа;

оператор 1; … — будь-які оператори мови С++.

У процесі виконання цього оператора спочатку обчислюється значення виразу **L**, потім це значення порівнюється (послідовно зверху донизу) зі значеннями константних виразів **к.в.1, …, к.в.N**. У випадку збігу значень **L** і одного з цих константних виразів та якщо наприкінці гілки розгалуження немає оператора **break**, виконуються всі оператори, починаючи з відповідної гілки. За наявності оператора **break** виконується тільки оператор, що знаходиться у відповідній гілці розгалуження, та керування передається оператору, який розташований за межами оператора **switch**. Якщо значення виразу **L** не збігається з жодним із значень константних виразів, то виконується гілка **default** і здійснюється вихід з оператора **switch**. У випадку, коли в операторі **switch** відсутня гілка **default** (вона необов'язкова) та значення **L** не збігається з жодним із значень константних виразів, відбувається вихід з оператора **switch** і виконується наступний за **switch** оператор.

Наведемо програму з використанням оператора **switch**, яка вирішує таку задачу: дано ціле число ***n = 1..3***, яке є номером функції. За значенням змінної ***n*** обчислити значення відповідної функції: ***1) -2x^2-4; 2) 5x+2; 3) 15-3x***, де ***x = 3***.

```
#include<iostream>
using namespace std;
int main()
{
        int n, x = 3;
        cout << "Input n = "; cin >> n;
        double f;
        switch (n)
        {
        case 1: f = -2 * x * x - 4; cout << "\n(1)f = " << f; break;
        case 2: f = 5 * x +2; cout << "\n(2)f = " << f; break;
        case 3: f = 15 -3* x; cout << "\n(3)f = " << f; break;
        default: cout << "\nError n! ";
        }
return 0;
}
```

Результати роботи програми в залежності від введеного значення:

```
Input n = 1          Input n = 2       Input n = 3       Input n = 5

(1)f = -22           (2)f = 17         (3)f = 6          Error n!
```

Та ж сама програма без **break** – виконуються всі оператори, починаючи з відповідної гілки до кінця оператору:

```
#include<iostream>
using namespace std;
int main()
{
        int n, x = 3;
        cout << "Input n = "; cin >> n;
        double f;
        switch (n)
        {
        case 1: f = -2 * x * x - 4; cout << "\n(1)f = " << f;
        case 2: f = 5 * x +2; cout << "\n(2)f = " << f;
        case 3: f = 15 -3* x; cout << "\n(3)f = " << f;
        default: cout << "\nError n! ";
        }
return 0;
}
```

Результати роботи в залежності від введеного значення::

```
Input n = 1          Input n = 2          Input n = 3          Input n = 5

(1)f = -22           (2)f = 17            (3)f = 6             Error n!
(2)f = 17            (3)f = 6             Error n!
(3)f = 6             Error n!
Error n!
```

Необхідно зважати на те, що гілка **default** не обов'язково має бути останньою, а може бути розташована будь-де в середині оператора:

```
#include<iostream>
using namespace std;
int main()
{
        int n, x = 3;
        cout << "Input n = "; cin >> n;
        double f;
        switch (n)
        {
        case 1: f = -2 * x * x - 4; cout << "\n(1)f = " << f;
        default: cout << "\nError n! ";
        case 2: f = 5 * x +2; cout << "\n(2)f = " << f;
        case 3: f = 15 -3* x; cout << "\n(3)f = " << f;
        }
return 0;
}
```

Результат:

```
Input n = 1

(1)f = -22
Error n!
(2)f = 17
(3)f = 6
```

5.2 Оператори циклу

Оператори циклу використовують для здійснення багаторазового повторення деякої послідовності дій. Кожен цикл складається з тіла циклу, тобто операторів, що виконуються декілька разів. Один прохід циклу називається *ітерацією*. У мові С++ існують три оператори циклу: **while**, **do...while**, **for.**

Оператор циклу з передумовою while виконується**,** якщо умова перевіряється до початку циклу, і має вигляд:

```
while (вираз-умова) оператор;,
```

де **оператор** — тіло циклу, що може бути представлено простим або складеним оператором.

Реалізується оператор **while** таким чином: якщо значення **вираз-умова** не дорівнює нулю («істина»), то виконується тіло циклу, а в протилежному випадку, тобто коли значення виразу дорівнює нулю («неправда»), — цикл не працює та керування передається наступному за циклом **while** оператору. Цикл з передумовою може не виконуватися жодного разу.

Як «вираз-умова» часто використовуються відношення. Наприклад, наступна програма обчислює суму перших *n* натуральних чисел:

```
#include <iostream>
using namespace std;

int main()
{
        int n, sum = 0;
        cout << "Input N : ";
        cin >> n;
        while (n > 0)
        {
                sum += n;
                n--;
        }
        cout << "Sum =\t" << sum << endl;
return 0;
}
```

Результат:

```
Input N : 23
Sum =   276
```

Якщо у виразі-умові необхідно порівняти покажчик з нульовим значенням (з порожнім покажчиком), то наступні записи оператора **while** рівнозначні:

```
while (point != NULL) ...
while (point) ...
while (point != 0) ...
```

Оператор **while** зручно застосовувати у випадках, коли кількість ітерацій заздалегідь не відома.

Оператор циклу з післяумовою do...while зазвичай застосовується у випадках, коли тіло циклу виконується хоча б один раз, і має таку форму запису:

```
do оператор while (вираз-умова);
```

У процесі виконання оператора **do ... while** спочатку здійснюється вхід до тіла циклу і виконується оператор, що становить тіло циклу (цей оператор може бути простим або складеним); далі перевіряється вираз і, якщо він правдивий («істина»), — цикл повторюється, а коли вираз помилковий («неправда») — здійснюється вихід з циклу.

Приклад обчислення суми перших *n* натуральних чисел з використанням **do... while** може бути реалізований таким чином:

```
#include <iostream>
using namespace std;
int main()
{
        int n, sum = 0;
        cout << "Input N : ";
        cin >> n;
        if (n < 0) //перевірка знаку числа
        cout << "The number is negative!";
         else
                do
        { sum += n;
          n--;
        } while (n > 0);
        cout << "Sum =\t" << sum << endl;
return 0;
}
```

Результати виконання:

```
Input N : 23
Sum =   276
```

```
Input N : -12
The number is negative!Sum =   0
```

Оператор циклу з параметром for має форму запису вигляду:

```
for ([вираз 1; вираз 2; вираз 3]) оператор;
```

де **вираз 1** — вираз ініціювання, що використовується для встановлення початкового значення параметра, це вираз присвоювання; **вираз 2** — вираз умови, що визначає умову повторення циклу; **вираз 3** — вираз ітерації, який визначає крок зміни параметра, що керує циклом.

Вирази **вираз 1**, **вираз 2** та **вираз 3** — необов'язкові параметри, які розділяються обов'язковими символами «**;**».

Схема виконання оператора **for**:

1. Обчислюється **вираз 1**, один раз перед входом у цикл.

2. Обчислюється **вираз 2** (перед кожним проходом циклу), якщо він відмінний від нуля – true (істина), то виконується тіло циклу, інакше (якщо вираз хибний) – цикл припиняється та управління передається оператору, наступному за оператором **for**.

3. Обчислюється **вираз 3** (модифікація даних після кожної ітерації циклу), перехід до пункту 2.

Істотно те, що перевірка умови виконується на початку циклу, а це означає, що тіло циклу може не виконатися жодного разу, якщо перший результат перевірки буде false.

Оскільки в операторі **for** перевірка виразу-умови відбувається перед циклом, то у випадку помилкової умови цикл може жодного разу не виконуватися.

Оператор **for** може використовувати декілька змінних, що керують циклом, а будь-які вирази можуть бути відсутніми, наприклад:

```
int = i;
    for (; i < 4; i++)…
```

або

```
    int n, y;
    for (int k = 0, n = 20; k != n; k++, n--)
    y = k * n;
```

Останній фрагмент має два вирази ініціювання та два вирази ітерації. Спочатку відбувається присвоювання значень змінним *k=0* і *n=20*, далі здійснюється порівняння *k != n* (*k* не дорівнює *n*) і, якщо ця умова має значення «істина», то виконуватиметься тіло циклу – обчислення *y*. Потім обчислюється вираз *k++* і *n--* та знову перевіряється порівняння *k!=n.* Цикл припиняє свою роботу, коли вираз *k != n* стає «неправдою».

Операторам циклів з параметром **for** треба віддати перевагу при організації циклів з лічильниками.

Приклад 5.1.

Обчислити значення функції $\mathbf{y = ax^2 - sin(x)}$, якщо $\mathbf{a = 10.5}$, $\mathbf{x \in [-1; 2]}$, $\mathbf{h_x = 0.5}$. Скласти програми з використанням різних операторів циклу.

Розглянемо перший варіант програми з використанням оператора циклу з передумовою **while**:

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
  double a = 10.5, x, y;
        x = -1;
        while (x <= 2)
        {
        y = a * pow(x, 2) - sin(x); // y = a * x * x - sin(x);
                cout << "x= " << x << "\ty= " << y << endl;
                x += 0.5;
        }
return 0;
}
```

Результати обчислення:

```
x= -1   y= 11.3415
x= -0.5 y= 3.10443
x= 0    y= 0
x= 0.5  y= 2.14557
x= 1    y= 9.65853
x= 1.5  y= 22.6275
x= 2    y= 41.0907
```

Другий варіант програми із застосуванням оператора циклу з післяумовою **do... while** має вигляд:

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
 double a = 10.5, x(-1), y;
        do
        {
                y = a * pow(x, 2) - sin(x);
                cout << "x= " << x << "\ty= " << y << endl;
                x += 0.5;
        }
        while (x <= 2);
return 0;
}
```

Результати обчислення:

```
x= -1   y= 11.3415
x= -0.5 y= 3.10443
x= 0    y= 0
x= 0.5  y= 2.14557
x= 1    y= 9.65853
x= 1.5  y= 22.6275
x= 2    y= 41.0907
```

Третій варіант програмної реалізації прикладу з використанням оператора циклу з параметром **for** має такий вигляд:

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
        double a = 10.5, x, y;
        for (x = -1.0; x < 2.001; x += 0.5)
        {
                y = a * pow(x, 2) - sin(x);
                cout << "x= " << x << "\ty= " << y << endl;
        }
return 0;
}
```

Результати обчислення:

```
x= -1   y= 11.3415
x= -0.5 y= 3.10443
x= 0    y= 0
x= 0.5  y= 2.14557
x= 1    y= 9.65853
x= 1.5  y= 22.6275
x= 2    y= 41.0907
```

Цикли можуть бути *вкладеними*. Цикл називається ***вкладеним***, якщо він розміщується всередині іншого циклу. На першому проході зовнішній цикл викликає внутрішній, який виконується до свого завершення, після чого управління передається в тіло зовнішнього циклу. На другому проході зовнішній цикл знову викликає внутрішній. І так до тих пір, доки не завершиться зовнішній цикл.

Приклад 5.2.

Обчислити значення функції **z** за різних значень **a** і **b**, якщо **x ∈[1; 4], h_x = 0.5; y ∈[4; 12], h_y = 2.**

$$z = \begin{cases} a^x + \sqrt{x+y}, & \text{якщо } x < 3; \\ ae^x + \ln(b+y), & \text{якщо } x \geq 3. \end{cases}$$

Розглянемо один з варіантів програмної реалізації цього прикладу, що відповідає алгоритму, зображеному на рисунку 5.1.

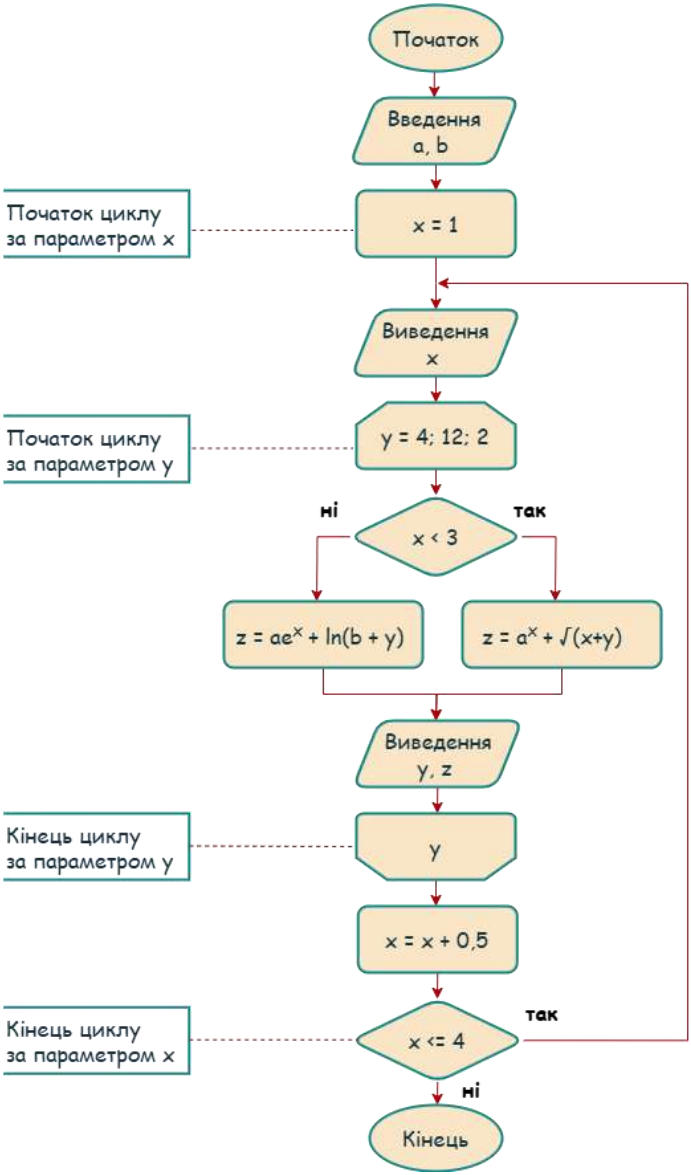


Рисунок 5.1 – Схема алгоритму прикладу 5.2

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{       double x, y, z;
        int a, b;
        cout << "Please input value for a = "; cin >> a;
        cout<<" and b = ";
        cin >> b;
        x = 1;
        do {    cout << "x = " << x << endl;
                for (y = 4; y <= 12; y += 2)
                {
                        if (x < 3) z = pow(a, x) + sqrt(x + y);
                        else z = a * exp(x) + log(b + y);
                        cout << "y = " << y << "\tz = " << z << endl;
                }
                x += 0.5;
            } while (x <= 4);
return 0;
}
```

Результати обчислення:

```
Please input value for a = 5
 and b = 7
x = 1
y = 4   z = 7.23607
y = 6   z = 7.64575
y = 8   z = 8
y = 10  z = 8.31662
y = 12  z = 8.60555
x = 1.5
y = 4   z = 13.5255
y = 6   z = 13.919
y = 8   z = 14.2625
y = 10  z = 14.5715
y = 12  z = 14.8546
x = 2
y = 4   z = 27.4495
y = 6   z = 27.8284
y = 8   z = 28.1623
y = 10  z = 28.4641
y = 12  z = 28.7417
x = 2.5
y = 4   z = 58.4512
y = 6   z = 58.8172
y = 8   z = 59.1421
y = 10  z = 59.4372
y = 12  z = 59.7096
```

```
x = 3
y = 4   z = 102.826
y = 6   z = 102.993
y = 8   z = 103.136
y = 10  z = 103.261
y = 12  z = 103.372
x = 3.5
y = 4   z = 167.975
y = 6   z = 168.142
y = 8   z = 168.285
y = 10  z = 168.41
y = 12  z = 168.522
x = 4
y = 4   z = 275.389
y = 6   z = 275.556
y = 8   z = 275.699
y = 10  z = 275.824
y = 12  z = 275.935
```

Функція ***z***, згідно з умовою, залежить від двох аргументів ***x*** і ***y***, що змінюються. У цьому випадку спочатку фіксується значення однієї змінної (наприклад, ***x***) і перебираються всі значення іншої (тобто ***y***), потім береться друге значення першої змінної та знову перебираються всі значення іншої, і так доти, доки не будуть використані всі значення першої змінної.

У програмі присутні два цикли: зовнішній цикл (**do...while**), у якому змінює свої значення змінна ***x***, і внутрішній (**for**), де перебираються всі значення змінної ***y***. Обидва ці цикли можна було б організувати за допомогою яких завгодно операторів циклу.

Процес розгалуження при обчисленні функції за першою або другою формулами здійснюється умовним оператором **if**.

5.3 Оператори керування

До операторів керування належить оператор безумовного переходу **goto** та оператори виходу з циклу.

Синтаксис **оператора безумовного переходу goto** має вигляд:

```
goto мітка;
```

де **мітка** — ім’я, після якого ставляться дві крапки**:**, область дії мітки — функція, в якій вона визначена.

Одним із способів використання **goto** є вихід з декількох рівнів вкладених циклів.

Наприклад:

```
    for (...) {
          for (...) {
              while (...) {
                  if (...) goto stop;
                  ...
                  }
                     }
                           }
stop:         cout << "Error in program!\n";
```

Знищення **goto** призведе до необхідності виконання додаткових перевірок. Простий оператор **break** тут не працює, оскільки він може вийти тільки з самого нижнього циклу.

Необхідно зауважити, що використання оператора безумовного переходу **goto** вважається негарним стилем програмування.

Мова С++ передбачає такі засоби виходу з циклу:

– оператор **break** призводить до негайного виходу з циклу (або з операторів **if** та **switch**), у якому він міститься, та переходу до наступного оператора програми, знехтувавши виконанням коду програми, що залишився в його тілі та перевіркою умовного виразу;

– оператор **continue** слугує для пропуску решти виконуваної частини операторів циклу, що міститься безпосередньо після його виклику. Якщо умовою циклу допускається нова ітерація, то вона виконується, в іншому випадку цикл завершується;

– функція **exit()** реалізує вихід з циклу із завершенням програми. Аргумент цієї функції — константа цілого типу, що дорівнює **0** у випадку сприятливого завершення циклу та відмінне від **0** значення в протилежному випадку. При використанні цієї функції необхідно включити в код програми директиву **#include <stdlib.h>**.

Порівняйте результати використання операторів **break** і **continue**:

```
#include<iostream>
using namespace std;
int main()
{for (int t = 0; t < 15; t++)
{
if (t == 10) break;
  cout << t << " ";
}
return 0;
}
```

```
#include<iostream>
using namespace std;
int main()
{for (int t = 0; t < 15; t++)
{
if (t == 10) continue;
        cout << t << " ";
}
return 0;
}
```

Результати:

0 1 2 3 4 5 6 7 8 9

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

5.4 Запитання та завдання

1. Які оператори реалізують розгалуження у програмі?
2. Як діє умовний оператор **if**? Наведіть приклади.
3. Як працює оператор-перемикач **switch**?
4. Які оператори циклу використовуються у С++?
5. Як працює оператор циклу **for**?
6. Поясніть на прикладах використання циклу з передумовою і циклу з післяумовою.
7. Які оператори керування застосовуються у мові С++?

Завдання.

Набути практичні навички використання операторів вибору, циклу та керування під час виконання завдань розділу вправ «Циклічні обчислювальні процеси».

6 МАСИВИ

6.1 Використання масивів

На практиці часто виникає необхідність в обробці даних у вигляді довільного набору значень, тобто масивів. **Масив** є кінцевою іменованою послідовністю величин одного типу, які розрізняються за порядковим номером. Опис масивів у програмі відрізняється від опису простої змінної наявністю після імені квадратних дужок [ ], в яких задається кількість елементів масиву (розмірність). ***В мові С++ нумерація елементів масиву починається з нуля***.

Розглянемо **одновимірні масиви**, оголошення яких допускає одну з таких форм запису (***n*** – **розмірність, кількість елементів**, **константне значення**):

```
<тип> ім'я [n];
<тип> ім'я [n] = {набір значень};
<тип> ім'я [ ] = {набір значень};
```

При оголошенні одновимірного масиву, коли масив відразу ініціюється, можна не вказувати його розмір. Якщо ж ініціювання не здійснюється під час оголошення масиву, то кількість індексів необхідно задати обов'язково константним виразом. Наприклад:

```
float m [6];
float m [6] = {3.4, 4.5, 5.6, 6.7, 8.9, 10.3};
float m [ ] = {3.45, 4.56, 5.67, 6.78};
```

Зрозуміло, що надалі ***кількість елементів змінити неможливо***. При оголошенні можна обнулити всі елементи, задавши нульове значення у фігурних дужках:

```
int mas [0] = {0};
```

За замовчуванням, якщо в оголошеному масиві ініціюється тільки декілька перших елементів, то його інші елементи ініціюються нулями. Так, у випадку, коли **float mas [10] = {2.2, 4.6};**, останні вісім елементів масиву одержать значення **0**.

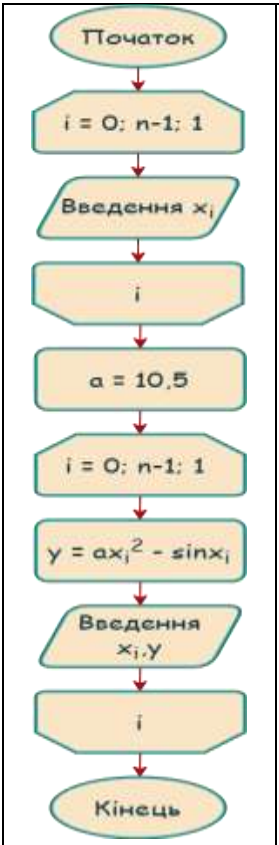
Проілюструємо використання одновимірних масивів на конкретних прикладах.

Приклад 6.1.

Обчислити функцію $\mathbf{y = ax_i^2 - \sin(x_i)}$, аргументи якої $\mathbf{x_i}$ — елементи одновимірного масиву, що мають значення:

$\mathbf{x_0 = 0.81; x_1 = 0.58; x_2 = 0.11; x_3 = 0.2; x_4 = 0.91; x_5 = 1.83}$.

Схему алгоритму та програмну реалізацію цієї задачі наведено на рисунку 6.1. Алгоритм передбачає введення значень одновимірного масиву ***x_i(i = 0…n-1), n = 6*** та подальше застосування їх для обчислення функції.




```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
const int  n = 6;
double x[n];    // опис масиву
double y, a = 10.5;
int i;
// введення масиву
cout<<"Enter the elements of the array:\n";
for (i = 0; i < n; i++)
{cout << "Enter " << i << " element  ";
        cin >> x[i];    }
// обчислення функції
for (i = 0; i < n; i++)
{ y = a * pow(x[i], 2) - sin(x[i]);
cout << "At value = " <<x[i];
cout <<"\ty = " << y << endl;}
return 0;
}
```

Результати роботи програми:

```
Enter the elements of the array:
Enter 0 element  0.81
Enter 1 element  0.58
Enter 2 element  0.11
Enter 3 element  0.2
Enter 4 element  0.91
Enter 5 element  1.93
At value = 0.81 y = 6.16476
At value = 0.58 y = 2.98418
At value = 0.11 y = 0.0172717
At value = 0.2  y = 0.221331
At value = 0.91 y = 7.90555
At value = 1.93 y = 38.1753
```

Рисунок 6.1 – Схема алгоритму та програма прикладу 6.1

У програмі спочатку описується масив дійсних значень: *double x[n].* Введення елементів масиву здійснюється у циклі, що реалізований оператором *for*. Цей цикл містить операцію потокового введення *cin >> x[i];*, перед якою знаходиться підказка *cout <<"x[" <<i<<"] = ";* для вказівки номера елемента *x[i]*. Особливість виконання операції введення *cin >> x[i];* полягає в тому, що: зустрівши її в програмі,

комп'ютер призупинить виконання програми, доки не буде введене значення елемента *x[i]* та натиснута клавіша *Enter*, після чого робота програми буде продовжена. Зазначена операція введення повторюється *n* разів для забезпечення введення всіх елементів масиву.

Оскільки в C++ індексація елементів масиву починається з нуля, то масив *double x[6] (n = 6)* із шести елементів включає індексовані елементи *x[0], x[1], x[2], ..., x[5]*. Програма використовує два цикли: один — для введення масиву, інший — для обчислення функції.

Операції введення елементів масиву та обчислення значень функції можна здійснити в одному циклі.

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
const int n = 6;
double x[n], y, a(10.5);
int i;
        for (i = 0; i < n; i++)
        {
            cin >> x[i];      // введення масиву
            y = a * x[i] * x[i] - sin(x[i]);
        // виведення результату
            cout<<" x["<<i<<"] = "<<x[i] <<"\t y = "<<y<<endl;
        }
return 0;
}
```

Результати обчислень з іншими значеннями x:

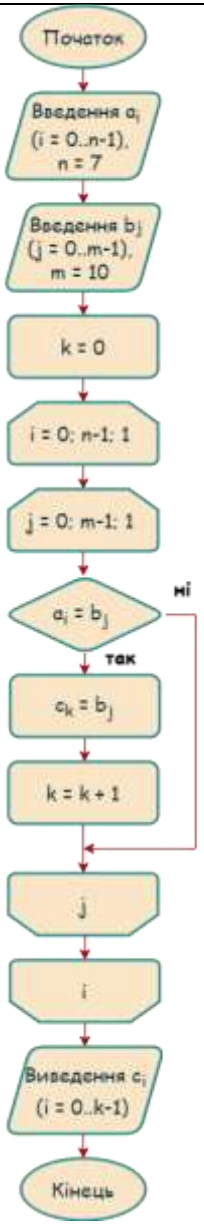
```
-0.81 2.13 -4.11 0.56 2.71 6.04
 x[0] = -0.81    y = 7.61334
 x[1] = 2.13     y = 46.7898
 x[2] = -4.11    y = 176.543
 x[3] = 0.56     y = 2.76161
 x[4] = 2.71     y = 76.6947
 x[5] = 6.04     y = 383.298
```

У цьому випадку передбачено введення елементів масиву в рядок і клавішу **Enter** потрібно натиснути в кінці процесу введення.

Приклад 6.2.

Сформувати масив c_k, який містить однакові елементи двох масивів a_i **(*i = 0...n-1*), *n = 7*** та b_j **(*j = 0...m-1*), *m = 10***. Масиви ***a*** і ***b*** не мають елементів, що повторюються.

Схему алгоритму та програмну реалізацію задачі наведено на рисунку 6.2.




```
#include<iostream>
using namespace std;
int main()
{
const int  n = 7, m = 10;
int a[n], b[m], c[m];    // опис масивів
int i, j, k;
// введення масивів
cout << " Input array A:" << endl;
for (i = 0; i < n; i++)
                cin >> a[i];
cout << " Input array B:" << endl;
for (j = 0; j < m; j++)
                cin >> b[j];
        k = 0;
for (i = 0; i < n; i++)
for (j = 0; j < m; j++)
if (a[i] == b[j])
{
c[k] = b[j];  k++;
} // c[k++]=b[j]
cout << " Rezult array C:" << endl;
for (i = 0; i < k; i++)
        cout << c[i] << "  ";
return 0;
}
```

Результати обчислень:

```
 Input array A:
6 12 2 55 7 9 33
 Input array B:
2 5 7 11 44 8 55 14 21 6
 Rezult array C:
6  2  55  7
```

Рисунок 6.2 – Схема алгоритму та програма прикладу 6.2

Цей приклад розв'язується з використанням вкладених циклів. Процес порівняння елементів відбувається немов у «три руки». Одна рука за параметром *i* вибирає елемент з масиву a_i, друга за параметром ***j*** вибирає елемент з масиву b_j, а третя за параметром ***k*** розташовує вибраний елемент у масив c_k.

Спочатку $i = 0$ – відбувається порівняння з a_0, а змінна ***j*** змінюється від *0* до *(m-1)*.

У циклі за параметром ***j*** кожний елемент b_j порівнюється з a_i доти, доки не знайдеться $a_i = b_j$ та не буде переглянутий весь масив b_j. Якщо $a_i = b_j$, то b_j заноситься до поточного елементу масиву c_k.

Далі повторюються аналогічні порівняння для ***i (i = 1, 2, …, n-1)***, тобто здійснюється порівняння елементів масиву b_j з наступним елементом масиву a_i.

Приклад 6.3.

За один перегляд масиву c_i ***(i = 0…n-1), n = 15*** визначити значення, а також положення максимального та мінімального його елементів та поміняти їх місцями.

Схему алгоритму розв'язання задачі та пояснення до неї наведено у прикладі 1.3 (рис. 1.3).

```
#include<iostream>
using namespace std;

int main()
{
const int n = 10;
float c[n]= {6.4, 1.5, 5.6, 3.7, 18.9, 50.3, 0.6, 44.5, 0.2, 8.9};
// опис масиву c[n] та його ініціалізація

float max, min;// максимальне (max) та мінімальне (min) значення
int imax, imin;  // індекси шуканих елементів
// виведення заданого масиву c[n]
cout << " ***** array c[n] ***** n= " << n << endl;
for (int i = 0; i < n; i++)
        cout << c[i] << "  ";
/*визначення максимального та мінімального елементів масиву та їхніх
індексів – imax, imin */
max = min = c[0];
imax = imin = 0;
for (int i = 1; i < n; i++)
{
if (c[i] > max)
        { max = c[i];
          imax = i; }
        else
             if (c[i] < min)
```

```
                    { min = c[i];  imin = i; }
}
// перестановка max и min елементів
        c[imin] = max;
        c[imax] = min;
// виведення max, min, imax, imin
        cout << "\n\t max= " << max << "   min= " << min << endl;
        cout << "\t imax= " << imax  << "   imin= " << imin << endl;
// виведення перетвореного масиву c[n]
        cout << " **** Rezult array ****" << endl;
for (int i = 0; i < n; i++)
        cout << c[i] << "  ";

return 0;
}
```

Результати обчислень:

```
***** array c[n] ***** n= 10
6.4  1.5  5.6  3.7  18.9  50.3  0.6  44.5  0.2  8.9
         max= 50.3   min= 0.2
         imax= 5   imin= 8
 **** Rezult array ****
6.4  1.5  5.6  3.7  18.9  0.2  0.6  44.5  50.3  8.9
```

Крім одновимірних масивів, тобто таких, де позиція елемента визначається за допомогою одного індексу, в практиці розв'язання задач часто застосовуються багатовимірні масиви. В них позиція елемента визначається записом декількох індексів.

Багатовимірний масив оголошується таким чином:

```
тип ім'я[розмір1][розмір2]...[розмірN];
```

Наприклад, за допомогою такого оголошення створюється тривимірний цілочисельний масив розміром 4x10x3:

```
int dimention[4][10][3];
```

Пам'ять, виділена для зберігання всіх елементів масиву, використовується протягом всього часу наявності масиву. Масиви з числом вимірювань, що перевищує три, використовуються нечасто, хоча б тому, що для їхнього зберігання потрібен великий об'єм пам'яті. Наприклад, зберігання елементів чотиривимірного символьного масиву розміром 10x6x9x4 займе 2 160 байтів.

Найбільш розповсюджені **двовимірні масиви** або матриці. Матриці є порядковим записом декількох одновимірних масивів. Індекси двовимірних масивів записуються в квадратних дужках і нумерація індексів також починається з нуля.

Елементи двовимірного масиву (матриці) визначаються іменем масиву та двома індексами: перший індекс означає номер рядка, інший – номер стовпця, на перетині яких розміщений елемент:

```
тип ім'я_масиву [n][m];
```

Наприклад, двовимірний масив цілих чисел *int a[3][4]*, що має три рядки та чотири стовпці, наведений на рисунку 6.3.

a[0][0]	**a[0][1]**	**a[0][2]**	**a[0][3]**
a[1][0]	**a[1][1]**	**a[1][2]**	**a[1][3]**
a[2][0]	**a[2][1]**	**a[2][2]**	**a[2][3]**

Рисунок 6.3 – Вигляд двовимірного масиву (матриці) *int a[3][4]*

А в пам'яті комп'ютера масив розташовується безперервно за рядками:

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[1][0]	a[1][1]	a[1][2]	…	a[2][2]	a[2][3]

Двовимірні (та багатовимірні) масиви можна оголошувати так:

```
int mas [2][5] ={ 1, 5, 3, 7, 4, 10, 11, 13, 14, 25} ;
int mas [ ][5] ={ 1, 5, 3, 7, 4, 10, 11, 13, 14, 25};
int mas [ ][5] ={ { 1,  5,  3,  7,  4 },
                  {10, 11, 13, 14, 25} }; ,
```

тобто масив задається або списком елементів у тому порядку, в якому вони розташовані у пам'яті, або подається як масив масивів, кожний з яких поміщається в свої фігурні дужки «**{}**». При оголошенні та одночасному ініціюванні багатовимірних масивів можна опускати кількість індексів тільки першого виміру. Якщо ініціювання не здійснюється під час оголошення масиву, то кількість індексів треба вказувати явно.

Для здійснення введення-виведення, а також для обробки елементів двовимірного масиву у програмі треба передбачати організацію двох циклів: один — для задання значень індексу рядків, інший — індексу стовпців.

Приклад 6.4.

Кожен елемент матриці ***M[3][4]*** збільшити на задане число.

```
#include<iostream>
using namespace std;
int main()
```

```
{const int n = 3, m = 4; // n та m – кількість рядків і стовпців
                         // матриці
        float M[n][m], z = 10;          // z – задане число
        int i, j;
        // введення елементів матриці та збільшення їх значень на z
                cout <<"*****Input matrix "<< endl;
                for (i = 0; i<n; i++)
                        for (j = 0; j<m; j++)
                {
                        cout <<" M ["<<  i <<"]"<<"["<<  j <<"]=";
                        cin >> M[i][j];
                        M[i][j] += z;
                }    // M [i][j] = M [i][j] + z;
        // виведення отриманої матриці в природньому вигляді
                cout <<"\n\n ***** Rezult matrix: ";
                for (i = 0; i<n; i++)
                {
                        cout << endl;
                        for (j = 0; j<m; j++)
                                cout << M[i][j] <<"  ";
                }
return 0;
}
```

Результати виконання програми:

```
*****Input matrix
 M [0][0]=4.5
 M [0][1]=6.7
 M [0][2]=4.8
 M [0][3]=23.6
 M [1][0]=5.7
 M [1][1]=3.7
 M [1][2]=2.9
 M [1][3]=6.1
 M [2][0]=1.2
 M [2][1]=4.5
 M [2][2]=4.6
 M [2][3]=2.7


 ***** Rezult matrix:
14.5  16.7  14.8  33.6
15.7  13.7  12.9  16.1
11.2  14.5  14.6  12.7
```

У програмі в ході опису матриці *float M[n][m];* вказується діапазон зміни двох індексів, перший з яких призначений для індексування рядків (***i***), другий — для індексування стовпців (***j***). Введення, обробка та виведення елементів матриці здійснюються за допомогою двох циклів, один з яких є вкладеним в інший. Це дозволяє при кожному

значенні змінної ***i*** перебирати всі значення змінної ***j***. Розглянута програма може бути скорочена шляхом об'єднання всіх трьох блоків циклу в один, але в такому випадку вона буде менш наочною.

Приклад 6.5.

Елементи головної та побічної діагоналей матриці ***C[4][4]*** поміняти місцями за відповідними рядками. Визначити максимальний елемент перетвореної матриці, а також номери рядка та стовпця, на перетині яких він знаходиться.

Схему алгоритму до прикладу наведено на рисунку 6.4.

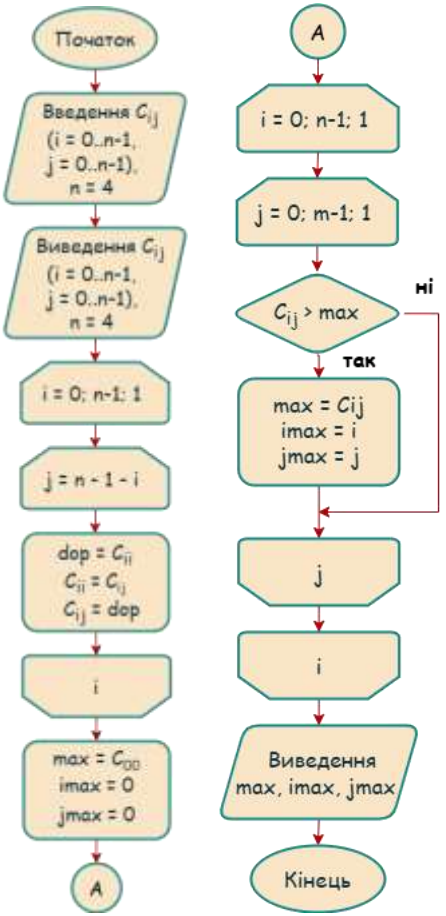


Рисунок 6.4 – Схема алгоритму прикладу 6.5

Програму розроблено з використанням алгоритму знаходження максимального елемента масиву та його індексів (приклад 1.3).

```
#include<iostream>
using namespace std;


int main()
{const int n = 4;
        int i, j, imax, jmax;
        float max, C[][n] = { { 3.6, 8.9, 1.9, 5.8 },
        { 8.8, 4.1, 1.2, 6.3 },
        { 2.5, 6.4, 0.1, 5.5 },
        { 8.8, 4.1, 1.2, 6.3 } };  // ініціалізація C[n][n]
        //  виведення початкової матриці в природньому вигляді
        cout << "  ***** matrix  C[n][n] ***** ";
        for (i = 0; i < n; i++)
        {
                cout << endl;
                for (j = 0; j < n; j++)
                        cout << C[i][j] << "  ";
        }

        // перестановка елементів головної та побічної діагоналей
        float dop;           //  допоміжна змінна для перестановки
        for (i = 0; i < n; i++) //for (i = 0, j=n-1; i < n; i++, j--)
        {
                j = n - 1 - i;          //      {  dop = C[i][i];
                dop = C[i][i];          //         C[i][i]= C[i][j];
                C[i][i] = C[i][j];      //         C[i][j]= dop;   }
                C[i][j] = dop;
        }

        // виведення перетвореної матриці
        cout << "\n\n  ***** REZULT matrix ***** ";
        for (i = 0; i < n; i++)
        {
                cout << endl;
                for (j = 0; j < n; j++)
                        cout << C[i][j] << "  ";
        }
  /* визначення max елементу матриці та його індексів – imax, jmax*/
        max = C[0][0];
        imax = jmax = 0;
        for (i = 0; i < n; i++)
                for (j = 0; j < n; j++)
                        if (C[i][j] > max)
                        {
                                max = C[i][j];
                                imax = i;    jmax = j;
                        }
```

```
cout << "\n\n max =  " << max;
cout << "   index i = " << imax << "   index j = " << jmax;
return 0;
}
```

Результати обчислень:

```
  ***** matrix  C[n][n] *****
3.6  8.9  1.9  5.8
8.8  4.1  1.2  6.3
2.5  6.4  0.1  5.5
8.8  4.1  1.2  6.3

  ***** REZULT matrix *****
5.8  8.9  1.9  3.6
8.8  1.2  4.1  6.3
2.5  0.1  6.4  5.5
6.3  4.1  1.2  8.8

 max =  8.9   index i = 0   index j = 1
```

6.2 Покажчики та масиви

Покажчики — це змінні, котрі містять адресу пам'яті, розподіленої для об'єкта відповідного типу. Усі змінні, розглянуті до цього, зберігали якісь значення (дані). Ці дані могли бути різних типів: символьного, цілого, дійсного тощо. При оголошенні змінної — покажчика необхідно вказати тип даних, адресу яких міститиме змінна, та ім'я покажчика з символом «*».

Формат опису простого покажчика має вигляд:

```
тип *ім'я;
```

де **тип** — тип значень, на який вказує покажчик;

ім'я — ім'я змінної-покажчика;

***** — операція над типом, що читається «покажчик на тип».

Наприклад:

```
int *pn; // покажчик на ціле значення;
float *pf1, *pf2; // два покажчики на дійсні значення.
```

Покажчики не прив'язують дані до якого-небудь визначеного імені змінної та можуть містити адреси будь-якого неіменованого значення. Існує адресна константа **NULL**, що означає порожню адресу.

Необхідно нагадати, що мова C++ налічує лише дві операції, які стосуються адрес змінних, а саме:

& — ***операція взяття адреси*** (адреса значення);

***** — ***операція розіменування*** (значення за адресою).

Операція взяття адреси **&** застосовується разом зі змінною та повертає адресу цієї змінної. Операція розіменування * використовується разом з покажчиками та вилучає значення, на яке вказує змінна-покажчик, розташована безпосередньо після символу «*»:

```
#include <iostream>
using namespace std;
int main()
{
    int a = 27;
    cout << "a = " << a;
    cout << "\n&a = " << &a;   //значення адреси пам'яті змінної a
    cout << "\n*&a = " << *&a; //значення ділянки пам'яті змінної a
return 0;
}
```

Результат:

```
a = 27
&a = 00000066A7AFF7C4
*&a = 27
```

Оголошення покажчиків можна здійснити одним з таких способів:

```
                    <тип> * ім’я;
            <тип> * ім’я = змінна-покажчик;
            <тип> * ім’я = &ім’я змінної; .
```

Наприклад:

```
int *ptx, b;
float y;
/* оголошені змінна-покажчик на ціле ptx та змінні b (цілого типу) і
y (дійсного типу) */
float *sp = &y;
/* покажчику на дійсний тип sp присвоюється адреса змінної y */
float *p = sp;
/*покажчику на дійсний тип p присвоюється значення (адреса-значення),
яке міститься в змінній sp, тобто адреса змінної y */.
```

При оголошенні покажчиків символ «*» може знаходитися перед ім’ям покажчика або відразу після оголошення типу покажчика та поширювати свою дію тільки на одну змінну-покажчик, перед якою він записаний:

```
long* pt; long * Uk; int *ki, x, h; //оголошення описів
```

За потреби для опису покажчика на комірку довільного типу замість ідентифікатора типу записується слово **void**, а саме:

```
void *p, *pt; //опис двох покажчиків на довільний тип даних.
```

Перед використанням покажчика у програмі його обов'язково необхідно ініціювати, іншими словами, необхідно присвоїти адресу якого-небудь даного, інакше можуть бути непередбачені результати.

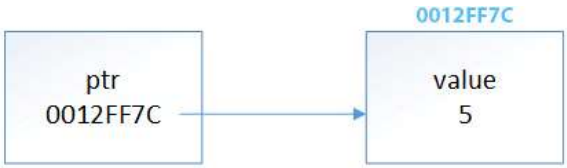
Для одержання доступу до значення змінної, адреса якої зберігається в покажчику, достатньо у відповідному операторі програми записати ім'я покажчика з символом * — здійснити операцію розіменування:

```
#include <iostream>
using namespace std;

int main()
{
   int value = 5;
   int* ptr = & value;              //ініціалізація покажчика
   cout << "value = " << value;     //значення змінної value
   cout << "\n&value = " << &value; //значення адреси пам'яті змінної
                                    // value
   cout << "\nptr = " << ptr;    //значення покажчика ptr
   cout << "\n*ptr = " << *ptr;  //значення, що зберігається за
                                 //адресою, на яку вказує ptr

return 0;
}
```

Результат:

```
value = 5
&value = 0012FF7C
ptr = 0012FF7C
*ptr = 5
```

Це можна проілюструвати таким чином:



Розглянемо ще один фрагмент програми з поясненнями:

```
int* p, *p1; //оголошені два покажчики на числа типу int - зберігають
//в собі адреси пам'яті інших змінних типу int
int x = 12, y = 5;
     //оголошені змінні x і y, змінні ініційовані
p = &y; // або p(&y) – покажчику p присвоєна адреса змінної y
```

Якщо для цього фрагмента програми записати оператор виведення у вигляді:

```
cout << "Address p = " << p << " Values at this address = " << *p;
```

то виведеться адреса комірки пам'яті, де записана змінна ***y*** і значення цієї змінної:

```
Address p = 00000010098FFBC4 Values at this address = 5
```

Використовуючи запис

```
x = *p;
```

отримаємо *x = 5*, тому що **p = y = 5*.

Змінити величину параметра *y* можна так:

```
y = 10;       // *p = 10
*p = *p+5;  //фактично y += 5; .
```

Остання операція означає збільшення значення змінної ***y*** (саме на неї вказує покажчик p) цілого типу на *5*, тобто *y = 15*.

При ініціюванні покажчиків їм можна присвоювати або адресу об'єкта (змінної), або адресу конкретного місця пам'яті (масиву), або число **0** (нуль),а саме:

```
int *pt = (char *) 0x00147; //присвоюється адреса комірки;
int *arrpt = new int [10];
//визначається початкова адреса розміщення динамічного масиву;
char *p = 0; // здійснюється ініціювання нулем.
```

Оскільки покажчики — це спеціальні змінні, то в операціях з іншими покажчиками вони можуть використовуватися без символу ***,** тобто без розкриття посилання, наприклад:

```
    float *pt1, *pt2, x=15, m[5];
    pt1 = &x;
    pt2 = pt1;
    pt1 = m;   // pt1 = &m[0];
```

де *m* — ім'я масиву, що розглядається як спеціальний покажчик-константа.

Приклад 6.6.

Написати ілюстративну програму з використанням покажчиків.

```
#include<iostream>
using namespace std;
int main()
{
        int x = 10;
        int* px(&x);      //   int *px  = &x;
```

```
        cout << "x = " << x << endl;
        cout << "*px = " << *px << endl;
        x *= 2;           //  x = x*2;
        cout << " New value *px  = " << *px << endl;
        *px += 2;         //  *px =*px + 2;
        cout << " Result *px, x = " << x << endl;
return 0;
}
```

Результат виконання програми:

```
x = 10
*px = 10
New value *px  = 20
Result *px, x = 22
```

Для змінної-покажчика існує своя адреса і тому будуть доцільними записи:

```
int *pt1, *pt2;
pt1 = (int*) &pt2;
//покажчику pt1 присвоюється адреса пам'яті, де розташована змінна pt2
```

Це має сенс у випадку, коли

```
int y, *pt1, *pt2 = &y;
pt1 = (int*) & pt2;
```

Обмеження на застосування операції взяття адреси «**&**»:

– не можна визначати адресу літеральної константи (оскільки для неї не виділяється комірка пам'яті), тобто такий запис, як *p = &345;* — неприпустимий;

– не можна визначати адресу результату арифметичного виразу, тобто запис *p=&(x + y);* теж неприпустимий.

Дозволені операції для змінних-покажчиків:

– операція розіменування *;

– операція взяття адреси &;

– операція присвоювання =;

– операції інкремент ++ і декремент --;

– операції додавання + і віднімання -;

– операції відношення (порівняння) покажчиків однакового типу: ==, !=,==, >, >=.

Можна визначити **константний покажчик**:

```
тип* const ідентифікатор = обов'язкова ініціалізація;
```

Подібно звичайним константам, константний покажчик має бути ініціалізований при оголошенні. Це означає, що він завжди вказуватиме на одну і ту ж саму адресу:

```
int num = 333;
int *const ptr = &num; //ptr завжди вказуватиме на адресу num
```

Покажчик на константу – це неконстантний вказівник, який вказує на константу:

```
const тип*ідентифікатор;
```

Покажчик на константну змінну не обов'язково має вказувати на константну змінну. Покажчик на константну змінну обробляє змінну як константу при отриманні доступу до неї незалежно від того, чи була ця змінна від самого початку визначена як const чи ні. Таким чином, наступне в порядку речей:

```
int num = 333;
const int *ptr = & num;     // ptr вказує на const int
num = 888;                  // змінна num не константа!
```

Але тут буде помилка:

```
int num = 333;
const int *ptr = & num // ptr вказує на const int
*ptr = 8; /* ERROR: ptr оброблює num як константу, тому зміна значення змінної num через ptr не допускається */
```

Покажчику на константне значення, який при цьому сам не є константним (він просто вказує на константне значення), можна присвоїти й інше значення:

```
int num1 = 777;
const int *ptr = & num1; // ptr вказує на const int
int num2 = 888;
ptr = & num2; //немає помилки, ptr тепер вказує на інший const int
```

У мові C++ масиви та покажчики зв'язані між собою: ***ім'я масиву визначається як покажчик-константа на початковий*** (***нульовий***) ***елемент масиву***. Так, наприклад, при оголошенні одновимірного масиву у вигляді *int mas [20];* його ім'я *mas* — покажчик на адресу початкового елемента масиву з індексом 0: *&mas[0]*.

Існує два способи доступу до елементів масиву:

1) з використанням індексу елемента масиву, наприклад, *mas[2]* або *mas[i];*

2) з використанням адресного виразу, тобто виразу з покажчиком на масив, наприклад, **(mas + 2)* або **(mas + i).*

Ім'я покажчика на масив можна записати так:

```
int mas [20];
int *ptr1;
ptr1 = mas;      // ptr1 = &mas[0];
```

тут вирази *&mas[0]* і *mas* еквівалентні.

Оскільки в комп'ютері для масивів завжди є суцільний блок комірок пам'яті, в яких розташовуються їхні елементи, то адресу наступного елемента *mas[1]* можна вказати шляхом збільшення покажчика на 1, а саме:

```
p = &mas[0];
p++;          // p = p + 1;
```

Таким чином, адреса *i*-го елемента визначається як *(p + i)*. При цьому з урахуванням типу масиву та відведеної кількості байтів для кожного його елемента автоматично виконується операція збільшення адреси, тобто:

```
адреса x[i] = адреса x[0] + i* sizeof (тип); .
```

Необхідно зауважити, що ***для покажчиків, які посилаються на елементи масивів різних типів, результат арифметичних операцій і операцій відношення невизначений***.

До двох покажчиків *p1* і *p2*, що вказують на елементи одного масиву, застосовують операції відношення: «==», «**!**=», «», «=», «**>**», «**>**=». При цьому значення покажчиків розглядаються як цілі числа, а результат порівняння дорівнює *0* («неправда») або *1* («істина»). Так, відношення вигляду *p1<p2* є «істина», якщо *p1* указує на більш ранній елемент, ніж *p2*. Будь-який покажчик можна порівняти з нулем.

В арифметиці з покажчиками можна використовувати адресу неіснуючого «наступного за масивом» елемента. До покажчиків можна додавати або віднімати від них цілу величину. В обох випадках результатом операції буде покажчик на вихідний тип, значення якого на вказане число елементів більше або менше вихідного. Тобто, якщо до покажчика *p* можна додати деяку цілу величину *n*, а саме: *p + n*, то цей вираз визначає ділянку об'єкта, що займає *n*-не місце після об'єкта, на який вказує *p*, при цьому *n* автоматично збільшується на коефіцієнт, що дорівнює відповідній довжині об'єкта. Наприклад, якщо *int* займає 4 байти, то цей коефіцієнт дорівнює чотирьом.

Допускається також операція віднімання покажчиків, що вказують на елементи одного масиву. Так, якщо *p1>p2*, то *(p2 - p1 + 1)* — це число елементів масиву від *p1* до *p2* включно.

Наведемо приклади програм роботи з покажчиками.

Приклад 6.7.

Обчислити середнє значення додатних елементів одновимірного масиву.

Розглянемо перший варіант програмної реалізації цієї задачі.

```
#include<iostream>
using namespace std;
int main()
{
        const int n = 10;
        float mas[n], s = 0;
        int i, k = 0; //k - кількість позитивних елементів
        cout << "Input 10 elements of array: " << endl;
        for (i = 0; i < n; i++)
                cin >> mas[i];
        for (i = 0; i < n; i++)
                if (mas[i] > 0)
                {
        s += mas[i];    // накопичення суми
        k++;            // знаходження кількості позитивних елементів
                }
        cout << "Average of positive elements = " << s / k << endl;
return 0;
}
```

Результати виконання програми:

```
Input 10 elements of array:
1.56 -4.78 6.5 -7.89 3.6 -9.45 7.4 -8.43 9.3 10.2
Average of positive elements = 6.42667
```

Використовуючи ім’я масиву як покажчик на початок масиву (перший елемент), можна навести другий варіант програми:

```
#include<iostream>
using namespace std;
int main()
{       const int n = 10;
        float mas[n], s = 0;
        int i, k = 0;           //k - кількість позитивних елементів
        cout << "Input 10 elements of array: " << endl;
        for (i = 0; i < n; i++)
        {
                cin >> *(mas + i);
                if (*(mas + i) > 0)
                {
                s += *(mas + i);        // накопичення суми
                k++;    //знаходження кількості позитивних елементів
                }
        }
        cout << "Average of positive elements = " << s / k << endl;
return 0;
}
```

Якщо описати покажчик і зв'язати його з масивом (адресувати на початок масиву), то з використанням арифметики покажчиків можна написати третій варіант цієї програми.

```
#include<iostream>
using namespace std;
int main()
{
const int n = 10;
int i, k(0);
float mas[n], s(0);
float* pm = mas;          //  pm=&mas[0];
        for (i = 0; i < n; i++)
        {cout << "Input " << i << " element of array " << endl;
         cin >> *pm++;
                if (mas[i] > 0) {
                        s += mas[i];   k++;
                                }
        }
cout << "\nAverage of positive elements = " << s / k << endl;
return 0;
}
```

У цій програмі для введення масиву застосований покажчик **pm*, а для роботи з масивом — ім'я масиву з індексом. Цей покажчик в операціях введення збільшує свою адресу *(pm++)* після введення чергового елемента масиву.

Наведемо четвертий варіант програмної реалізації прикладу:

```
#include<iostream>
using namespace std;
int main()
{
        const int n = 10;
        int i, k(0);
        float mas[n], s(0);
        float* pm = mas;          //  pm=&mas[0];
        for (i = 0; i < n; i++)
        {cout << "Input " << i << " element of array " << endl;
         cin >> *pm;
               if (*pm > 0)
                    { s += *pm;
                      k++;
                      pm++; }
        }
cout << "\nAverage of positive elements = " << s / k << endl;
return 0;
}
```

6.3 Масиви покажчиків

Подібно до інших змінних, покажчики можна групувати в масиви, кожен з елементів яких містить адресу рядка масиву даних у пам'яті. Такий спосіб дозволяє зберігати дані з «рваними» краями, наприклад, деяку текстову інформацію (див. рис. 6.5). Масив з «рваними» краями схожий на двовимірну таблицю, рядки якої можуть мати різну довжину. Використання масиву покажчиків (*char *pib[ ]*) для збереження рядків дозволяє заощаджувати пам'ять, а процес обробки рядків виконується значно швидше, бо змінюються тільки покажчики, а не вміст рядків.

Приклад 6.8.

Заданий масив з «рваними» краями (рис. 6.5). Навести програмну реалізацію виведення такої інформації з використанням масиву покажчиків.

Рисунок 6.5 – Приклад масиву з «рваними» краями

```
#include<iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "ukr");//для виведення українського шрифту
        const char*pib[] = {"Рильський", "Стус", "Андрухович",
"Шевченко", "Франко", "Жадан"};  // ініціалізація масиву покажчиків
        int rd;
        for (rd = 0; rd < 6; rd++)
        cout << " рядок " << rd << " =  " << *(pib + rd) << endl;
return 0;}
```

Результати виконання програми:

```
рядок 0 =  Рильський
рядок 1 =  Стус
рядок 2 =  Андрухович
рядок 3 =  Шевченко
рядок 4 =  Франко
рядок 5 =  Жадан
```

На рисунку 6.5 показано зв'язок масиву покажчиків *char * pib [ ]* з відповідними текстовими рядками.

Особливістю масиву покажчиків є те, що кожний з його елементів може вказувати на масив довільної довжини. Існує можливість записати двовимірний масив чисел і як матрицю, і як одновимірний масив покажчиків:

```
int matr[5][7];
```

або

```
int *pmt[5]; .
```

При цьому двовимірний масив розглядається як одновимірний масив рядків, кожен елемент якого — це теж масив стовпців, тобто масив масивів, тому індексування елементів матриці записується у вигляді *mas [i][j]*.

Звернення до елемента *mas[i][j]* може здійснюватися так:

```
*(pm[i] + j) або *(*(pm + i) + j) .
```

Приклад 6.9.

Сформувати матрицю цілих чисел ***C[5][5]***, елементи якої обчислюються за формулою $C_{ij} = i+j$. Підрахувати добуток елементів, розташованих нижче побічної діагоналі, та обнулити ці елементи. Вивести на екран елементи, розташовані в трикутниках нижче головної та вище побічної діагоналей.

Перший варіант програмної реалізації даної задачі передбачає, що матриця описується явним способом і робота ведеться з її елементами.

```
#include<iostream>
using namespace std;
const int n = 5;
int main()
{setlocale(LC_ALL, "ukr");//для українського шрифту
int i, j, C[n][n];
int Dob;   // Dob – змінна для обчислення добутку
cout << " --- Матриця  C[n][n]: \n"; // формування матриці C[n][n]
        for (i = 0; i < n; i++)
        {   for (j = 0; j < n; j++)
                {       C[i][j] = i + j;
                        cout << C[i][j] << " ";
                }
                cout << endl;
        }
    /* обчислення добутку (Dob) елементів нижче побічної діагоналі
                    та їх подальше обнуління */
        Dob = 1;     // початкове значення
        for (i = 0; i < n; i++)
         for (j = n - i; j < n; j++) //або for (j=n-1;j>n-i-1;j--)
```

```
                {       Dob *= C[i][j];
                        C[i][j] = 0;
                }
        //виведення на екран отриманих результатів
cout << "\n Добуток нижче побічної діагоналі = " << Dob << endl;
cout << "\n --- Перетворена матриця \n";
for (i = 0; i < n; i++)
        {for (j = 0; j < n; j++)
                 cout << C[i][j] << " ";
                 cout << endl;
        }
        //виведення елементів нижче головної діагоналі
        cout << "\n Елементи нижче головної діагоналі ";
        for (i = 0; i < n; i++)
        {       for (j = 0; j < n; j++)
                 if (j < i) cout << C[i][j] << "  ";
                        else cout << "  ";
                cout << endl;
        }
        //виведення елементів вище побічної діагоналі
cout << "\n Елементи вище побічної діагоналі \n";
        for (i = 0; i < n; i++)
        {   for (j = 0; j < n; j++)
                        if (j < n - 1 - i)
                             cout << C[i][j] << "  ";
                        else cout << "  ";
                cout << endl;
        }
return 0;
}
```

Результати обчислень:

```
--- Матриця  C[n][n]:
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

 Добуток нижче побічної діагоналі = 52920000

 --- Перетворена матриця
0 1 2 3 4
1 2 3 4 0
2 3 4 0 0
3 4 0 0 0
4 0 0 0 0

 Елементи нижче головної діагоналі
1
```

```
2  3
3  4  0
4  0  0  0

 Елементи вище побічної діагоналі
0  1  2  3
1  2  3
2  3
3
```

Другий варіант програмної реалізації використовує масив покажчиків:

```
#include<iostream>
using namespace std;
const int n = 5;
int main()
{
        setlocale(LC_ALL, "ukr");//для українського шрифту
        int i, j, Dob, C[n][n], * pm[n];
        // ініціалізація масиву покажчиків
        for (i = 0; i < n; i++)
                pm[i] = &C[i][0];
        // формування матриці C[n][n]
        cout << " Матриця  C[n][n]" << endl;
        for (i = 0; i < n; i++)
                for (j = 0; j < n; j++)  *(pm[i] + j) = i + j;
        // виведення матриці
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++)
                        cout << *(pm[i] + j) <<" ";
                cout << endl;
        }
/* обчислення добутку (Dob) елементів нижче побічної діагоналі та їх
подальше обнуління */
        Dob = 1;
        for (i = 0; i < n; i++)
                for (j = n - i; j < n; j++)
                {
                        Dob *= *(pm[i] + j);
                        *(pm[i] + j) = 0;
                }
        // виведення на екран отриманих результатів
cout << "\n Добуток нижче побічної діагоналі = " << Dob << endl;
cout << "\n Перетворена матриця " << endl;
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++)  cout << *(pm[i] + j) << " ";
                cout << endl;
        }
        // виведення елементів нижче головної діагоналі
```

```
        cout << "\n Елементи нижче головної діагоналі";
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++)
                        if (j < i) cout << *(pm[i] + j) << "   ";
                        else cout << "   ";
                cout << endl;
        }
        cout << endl;
        // виведення елементів вище побічної діагоналі
        cout << endl << " Елементи вище побічної діагоналі " << endl;
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n; j++)
                        if (j < n - 1 - i)  cout << *(pm[i] + j) << " ";
                        else cout << "   ";
                cout << endl;
        }
return 0;
}
```

Ім'я двовимірної матриці є покажчиком-константою на масив покажчиків-констант, кожний з елементів якого вказує на початок відповідного рядка матриці. Наприклад, для матриці **matr[2][2]** маємо:

matr [0] — покажчик-константа на нульовий рядок матриці;
matr [1] — покажчик-константа на перший рядок матриці;
matr [2] — покажчик-константа на другий рядок матриці;
тобто

```
matr [0] == &matr [0][0];
matr [1] == &matr [1][0];
matr [2] == &matr [2][0]; .
```

Виведення матриці можна реалізувати з використанням одного з наведених нижче операторів, наприклад:

```
cout <<  matr [i][j];
cout <<*(matr [i] + j);
cout <<*(*(matr + i) + j); .
```

Приклад 6.10.

Задана матриця a_{ij}***(i = 0...n-1, j =0...m-1) n = 3, m = 4,*** необхідно її парні елементи переписати до масиву ***b***, а непарні — до масиву ***c***.

Алгоритм розв'язання цієї задачі наведено у прикладі 1.6 (рис. 1.6).

```
#include<iostream>
using namespace std;
int main()
{
        setlocale(LC_ALL, "ukr");
        const int n = 3, m = 4;
        int a[n][m], b[m * n], c[m * n], i, j, kc = 0, kn = 0;
// kc та kn – лічильники підрахунку парних і непарних елементів
// введення початкової матриці a[n][m]
                cout << " Введення матриці a[3][4] " << endl;
                for (i = 0; i < n; i++)
                        for (j = 0; j < m; j++)  cin >> *(*(a + i) + j);
// формування масивів b[ ] та c[ ]
                for (i = 0; i < n; i++)
                for (j = 0; j < m; j++)
                if (*(*(a + i) + j) % 2 == 0)*(b + kc++) = *(*(a + i) + j);
                                else *(c + kn++) = *(*(a + i) + j);
// виведення масиву b[ ] – парних елементів
                cout << endl << "Масив парних елементів b[ ]: " << endl;
                for (i = 0; i < kc; i++)   cout << *(b + i) << " ";
                cout << endl;
// виведення масиву c[ ] – непарних елементів
                cout << endl << "Масив непарних елементів c[ ]: " << endl;
                for (i = 0; i < kn; i++)   cout << *(c + i) << " ";
                cout << endl;
// виведення початкової матриці a[n][m] в природньому вигляді
                cout << endl << "Початкова матриця";
                for (i = 0; i < n; i++)
                {cout << endl;
                for (j = 0; j < m; j++)   cout << *(*(a + i) + j) << " ";
                }
return 0;
}
```

Результати обчислень:

```
 Введення матриці a[3][4]
8 2 4 -1
6 1 0 5
2 9 3 -1

Масив парних елементів b[ ]:
8 2 4 6 0 2

Масив непарних елементів c[ ]:
-1 1 5 9 3 -1

Початкова матриця
8 2 4 -1
6 1 0 5
2 9 3 -1
```

В C++ можна описати змінну, що має **тип «покажчик на покажчик»**. Ознакою такого типу є повторення символу «*****» в ході опису змінної, тобто *int **pmt;* .

Під час опису покажчик на покажчик можна ініціювати:

```
#include<iostream>
using namespace std;
int main()
{       int x = 20;
        int* px1 = &x;
        int** px2 = &px1;
        int*** px3 = &px2;
        cout << &x << endl;
        cout<< &px1<<"  "<<px1<<endl;
        cout << &px2 << "  "<<px2<<endl;
        cout << &px3 << "  "<<px3<<endl;
        cout << ***px3;
return 0;
}
```

Результат виконання:

```
0000001F2AD2F9E4
0000001F2AD2FA08  0000001F2AD2F9E4
0000001F2AD2FA28  0000001F2AD2FA08
0000001F2AD2FA48  0000001F2AD2FA28
20
```

Таким чином, до змінної *x* можна дістатися різними способами: **px1, **px2, ***px3*.

6.4 Сортування масивів

Сортування масиву — один з найбільш розповсюджених процесів обробки даних. Завдяки йому здійснюється розміщення об'єктів у визначеному порядку, наприклад, чисел за зростанням або за спаданням їхніх значень, прізвищ у алфавітному порядку тощо. Існують різні методи сортування [3, 10], серед них— обмінне сортування (метод «бульбашки», «шейкер-сортування»), сортування вибором, сортування вставками, швидке сортування, сортування Шелла, пірамідальне сортування, сортування обчисленням адреси, сортування порозрядним групуванням тощо. Ці методи відрізняються швидкістю отримання результату, складністю та універсальністю.

Розглянемо три методи сортування: обміном, вибором і вставками. Названі методи легко описуються у формі чітких алгоритмів і передбачають нескладну програмну реалізацію, крім того вони цікаві тим, що моделюють природну поведінку людини, яка здійснює сортування вручну. З іншого боку, вказані методи не досить ефективні

та використовуються у випадках, коли необхідно відсортувати масиви невеликого розміру.

Обмінне сортування проілюструємо простим сортуванням обміном — **методом «бульбашки»**, який здійснюється шляхом перестановки елементів за визначеним правилом. У загальному випадку алгоритм сортування за цим методом наведений у прикладі 1.4. Розглянемо його більш детально.

Нагадаємо головні складові методу «бульбашки»:

1) крок сортування містить перегляд елементів масиву з початку до кінця, при цьому розглядаються пари сусідніх елементів;

2) елементи деякої пари міняються місцями у випадку, коли їх послідовність розташування не відповідає умові сортування (за зростанням або за спаданням).

Приклад 6.11.

Упорядкувати за зростанням методом «бульбашки» масив x_i $(i = 0...n-1)$, $n = 5$, що має значення:

$$x_0 = 25;\ x_1 = 37;\ x_2 = 0;\ x_3 = 10;\ x_4 = 2.$$

Необхідно зауважити, що ***жоден метод сортування не досягає результату за один перегляд масиву, для цього застосовується, як правило, цикл у циклі***.

В алгоритмі сортування за методом «бульбашки» (рис. 6.6) з цією метою використано зовнішній цикл (цикл кроків сортування) за параметром ***k*** та внутрішній цикл (цикл порівнянь та перестановок) за параметром ***i***. Оскільки кількість кроків сортування має бути ***n-1***, то параметр ***k*** зовнішнього циклу змінюється від ***1*** до ***n-1*** включно. На кожному кроці сортування в циклі за параметром ***i*** відбувається порівняння пар сусідніх елементів і їхня перестановка у випадку, коли пара розташована не за зростанням. Параметр ***i*** відповідає номеру елемента масиву.

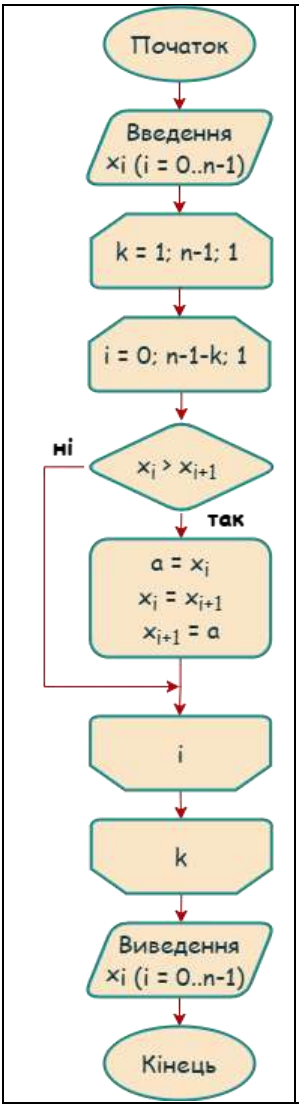
Перший крок сортування (**k= 1**) здійснюється так:

– послідовно порівнюються пари сусідніх елементів («25» – «37») і, якщо перший елемент більше за другий, вони міняються місцями, тобто на друге місце, як пухирець, «спливає» більший з двох елементів (у даному випадку елементи залишаються на своїх місцях, елемент, що «спливає», виділений жирним шрифтом):

25	**37**	0	10	2

– потім другий елемент («37»), більший з двох, порівнюється з третім елементом («0»), і на третє місце «спливає» більший з трьох («37»):

25	0	**37**	10	2




```
#include<iostream>
using namespace std;
int main()
{
const int n = 5;
// ініціалізація масиву

int x[n] = { 25, 37, 0, 10, 2 };
int a;    /* a – робоча змінна для
перестановки місцями двох елементів*/

int i, k;
// виведення на екран початкового масиву
cout << "\n Array  x[5]: \n";
    for (i =0; i < n; i++)
    cout << x[i] << "  ";

// сортування за зростанням
for (k = 1; k < n; k++)  // цикл кроків
                         // сортування
for (i = 0; i < n - k; i++)   /* цикл
порівняння елементів та їх перестановки */

     if (x[i] > x[i + 1])
        { a = x[i];
         x[i] = x[i + 1];
         x[i + 1] = a;
        }
cout<<"\n Result of sorting the array:\n";
   for (i = 0; i < n; i++)
     cout << x[i] << "  ";
return 0;
}
```

Результат:

```
 Array  x[5]:
25  37  0  10  2
 Result of sorting the array:
0  2  10  25  37
```

Рисунок 6.6 – Схема алгоритму та програма ***прикладу 6.11*** (метод «бульбашки»)

– далі третій елемент («37») порівнюється з четвертим елементом («10»):

25	0	10	**37**	2

– перегляд продовжується до кінця масиву, та найбільший елемент «спливе» та займе останнє місце у масиві:

25	0	10	2	**37**

Оскільки найбільший елемент займає своє місце після першого кроку сортування, то у наступному кроці сортування він участі не бере. Тобто при кожному наступному сортуванні кількість пар, що порівнюються, зменшується на 1, і параметр ***i*** внутрішнього циклу сортування залежить від параметра ***k*** циклу кроків сортування та дорівнює ***n·(k-1)***. Перестановка місцями двох елементів здійснюється за допомогою робочої змінної ***a***.

Для того, щоб серед елементів, котрі залишилися, знову «сплив» найбільший, необхідно провести аналогічні операції, тобто здійснити наступні кроки сортування.

Другий крок сортування (**k** = **2**), у результаті якого «спливає» вправо на передостаннє місце наступний найбільший елемент, дає таке розміщення елементів:

0	10	2	**25**	**37**

Третій крок сортування (**k** = **3**) дозволяє розташувати наступний за величиною елемент третім справа:

0	2	**10**	**25**	**37**

І далі останній ***четвертий крок сортування*** (**k** = **4**) дає результат:

0	**2**	**10**	**25**	**37**

Сортування масивів за методом «бульбашки» є найменш ефективним, середня кількість порівнянь дорівнює $(n^2 - n)/2$. Незважаючи на це, метод залишається одним з найпопулярніших завдяки простоті реалізації.

Особливості застосування методу «бульбашки»:

– алгоритми сортування, як за зростанням, так і за спаданням елементів, майже однакові, відрізняються вони тільки знаками порівняння, тобто, наприклад, замість $(x_i > x_{i+1})$ **необхідно** записати прямо протилежне: $(x_i < x_{i+1})$;

– присутність однакових елементів у масиві не додає проблем: у момент порівняння обидва елементи залишаються на своїх місцях, а потім послідовно зміщуються по ряду та займають своє остаточне місце на сусідніх позиціях.

Наведемо другий варіант програмної реалізації прикладу, в якому ім'я масиву використано як покажчик на його перший елемент:

```
#include<iostream>
using namespace std;
int main()
{const int  n = 5;
int x[n], i, k;
int a;   // a – робоча змінна для перестановки місцями двох елементів

//------- введення початкового масиву
        for (i = 0; i < n; i++)  cin >> *(x + i);
//------- виведення на екран початкового масиву
        cout << "\n Array  x[5]:\n";
        for (i = 0; i < n; i++)  cout << *(x + i) << "  ";
//------- сортування за зростанням
for (k = 1; k < n; k++)      //цикл кроків сортування
for (i = 0; i < n - k; i++)  //цикл порівняння елементів та їх
                             //перестановки
        if (*(x + i) > *(x + i + 1))
        {
          a = *(x + i);
          *(x + i) = *(x + i + 1);
          *(x + i + 1) = a;
        }
cout << "\n Result of sorting the array: " << endl;
        for (i = 0; i < n; i++) cout << *(x + i) << "  ";
return 0;
}
```

В цій реалізації елементи масиву вносяться користувачем. Результат:

```
Input array  x[5]:
12 -34 81 45 23

 Array  x[5]:
12  -34  81  45  23
 Result of sorting the array:
-34  12  23  45  81
```

Приклад 6.12.

Для матриці ***matr[5][6]*** знайти суми елементів кожного рядка та записати їх в одновимірний масив. Отриманий масив відсортувати за зростанням методом «бульбашки».

```
#include<iostream>
using namespace std;
int main()
{
int matr[5][6], mas[5] = {0};      // mas[] – масив сум рядків
int  i, j, temp;
```

```
// ---------введення матриці matr(5,6)
cout << "Input matrix [5][6] \n";
for (i = 0; i < 5; i++)
   for (j = 0; j < 6; j++)
   cin >> matr[i][j];
//--------- формування масиву сум елементів рядків
for (i = 0; i < 5; i++)
// знаходження суми елементів рядка
    for (j = 0; j < 6; j++)
        mas[i] += matr[i][j]; // запис суми рядка в масив
cout << "\nArray of sums of row elements" << endl;
for (i = 0; i < 5; i++)
        cout << mas[i] << "  ";
//------ сортування масиву сум за зростанням
for (i = 1; i < 5; i++)      //цикл кроків сортування
  for (j = 0; j < 5 - i; j++)//цикл порівняння та перестановки
                             //елементів

                if (mas[j] > mas[j + 1])
                {
                temp = mas[j];   // temp – для перестановки елементів
                mas[j] = mas[j + 1];
                mas[j + 1] = temp;
                }
cout << "\nSorted array \n";
for (i = 0; i < 5; i++)  cout << mas[i] << "  ";
return 0;
}
```

Результати обчислень:

```
Input matrix [5][6]
5 5 5 5 5 5
4 4 4 4 4 4
3 3 3 3 3 3
2 2 2 2 2 2
1 1 1 1 1 1

Array of sums of row elements
30  24  18  12  6
Sorted array
6  12  18  24  30
```

У програмі використаний типовий прийом алгоритмізації — формування робочого масиву mas_i, що запам'ятовує суми елементів кожного рядка матриці. Застосування такого прийому докладно пояснюється у прикладі 1.2 та прикладі 1.5. Кількість елементів створеного масиву дорівнює кількості рядків матриці. Для сортування отриманого масиву за зростанням використано сортування методом «бульбашки».

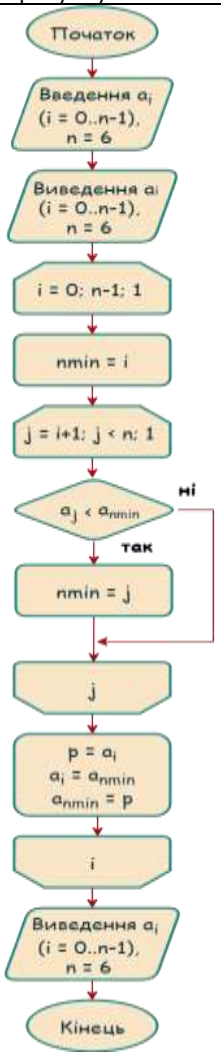
Сортування за методом вибору розглянемо на прикладі.

Приклад 6.13.

Упорядкувати за зростанням методом вибору масив, що має такі значення:

$$x_0 = 20;\ x_1 = 1;\ x_2 = 30;\ x_3 = 2;\ x_4 = 7,\ x_5 = 5.$$

Схему алгоритму та програму реалізації даної задачі наведено на рисунку 6.7.




```
#include<iostream>
using namespace std;
int main()
{
const int n = 6;
int a[n] = {20, 1, 30, 2, 7, 5};
int nmin, i, j, p;
// виведення масиву на екран
cout << "Array a[6]: " << endl;
      for (i = 0; i < n; i++)
      cout << a[i] << "  ";
cout << endl;

// цикл кроків сортування
for (i = 0; i < n - 1; i++)
{
  nmin = i;
/* цикл визначення номеру мінімального
елементу*/

for (j = i + 1; j < n; j++)
        if (a[j] < a[nmin])
         nmin = j;

/*перестановка мінімального елементу з тим,
що розглядається */
        p = a[i];
/* p – для перестановки елементів*/
        a[i] = a[nmin];
        a[nmin] = p;
        }
// виведення відсортованого масиву
cout << "\nSorted array:" << endl;
for (i = 0; i < n; i++)
        cout << a[i] << "  ";
return 0;
}
```

Результат обчислень:

```
Array a[6]:
20  1  30  2  7  5

Sorted array:
1  2  5  7  20  30
```

Рисунок 6.7 – Схема алгоритму та програма прикладу 6.13 (метод вибору)

Процес сортування за зростанням здійснюється за кроками. Позначимо номер кроку сортування параметром ***i***. На кожному кроці знаходиться найменший елемент, що міняється місцями з елементом, номер якого збігається з номером кроку ***i***.

Нульовий крок сортування (**i = 0**):

– у процесі розгляду елементів масиву, починаючи з першого, знаходять найменший елемент («1») і розташовують його на місце першого елемента, а перший («20») — на місце мінімального. У результаті найменший елемент масиву потрапляє на нульову позицію, тобто на перше місце зліва (підкреслено елементи, що переставляються):

Перший крок сортування (**i= 1**):

– наступний найменший елемент («2») знаходять у частині масиву, що починається з першої позиції. Він міняється місцями з другим елементом («20»), тобто другий за значенням елемент («2») розташується на першій позиції, а саме на другому місці зліва:

Другий крок сортування (**i= 2**):

– третій за значенням елемент («5») знайдемо у масиві, починаючи з другої позиції («30»), та поміняємо його місцем з елементом масиву, що розташувався на другій позиції:

Подальші кроки сортування дають такі перетворення:

третій крок сортування (**i= 3**):

четвертий крок сортування (**i = 4**):

Як результат маємо відсортований за зростанням масив:

Загальна кількість дій алгоритму сортування за методом вибору дорівнює $(n^2/4+ 3n)$ операціям.

Сортування за методом вставки полягає в тому, що на кожному кроці відбувається вставка елемента у відсортований масив. Розглянемо цей метод на конкретному прикладі.

Приклад 6.14.

Упорядкувати за зростанням (за спаданням) методом вставки масив a_i $(i = 0...n-1), n = 6$, що має такі значення:

$$a_0 = 4;\ a_1 = 13;\ a_2 = 3;\ a_3 = 6;\ a_4 = 8,\ a_5 = 24.$$

Схему алгоритму та програму реалізації цієї задачі наведено на рисунку 6.8.

Сутність алгоритму сортування за методом вставки:

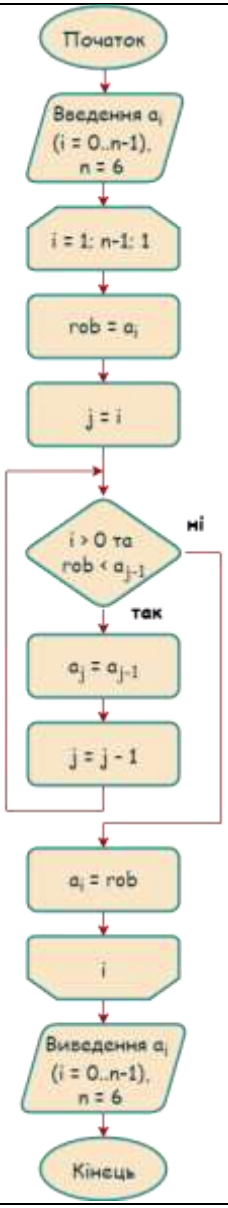
– якщо $j\ (a_0...a_{j1})$ елементів масиву a відсортовані за зростанням, а елемент a_j має довільне значення, потрібно порівнювати цей елемент по черзі з елементами $a_{j1}, a_{j2}, ...,$ доки для a_j не знайдеться місце у відсортованому масиві. При цьому всі розглянуті елементи $a_{j1}, a_{j2}, ...,$ що будуть за значенням більші, ніж a_j, мають переміститися на одну позицію вправо;

– у випадку, коли елемент-вставка a_j виявиться більше за значенням, ніж a_{j1}, елемент a_j залишиться на своєму місці. Якщо a_j буде менше всіх елементів $a_0...a_{j1}$, то всі елементи мають бути переміщені на одну позицію вправо, а a_j займе місце $j=0$. Щоб при переміщенні вправо не загубити значення елемента-вставки a_j, потрібно завчасно зберегти його в робочій змінній (***rob***).

У програмі параметр ***i*** зовнішнього циклу відповідає за номер елемента-вставки, тому на першому кроці ($i = 1$) вважаємо, що відсортований масив, до якого вставляється елемент $rob = a_1$, складається тільки з одного елемента, на наступному кроці ($i == 2$) маємо відсортований масив з двох елементів, а вставляємо до нього елемент a_2 і так далі.

У внутрішньому циклі програми за параметром ***j*** відбувається порівняння елемента-вставки ***rob*** з елементами відсортованого масиву і переміщення цих елементів вправо, якщо за значенням вони більші, ніж ***rob***. Безпосередньо вставка відбувається у зовнішньому циклі: $a_j = rob$.

З розглянутих методів сортування цей метод — найефективніший, середня кількість операцій приблизно дорівнює $n^2/4$.




```
#include<iostream>
using namespace std;
int main()
{
const int n = 6;
int a[n], i, j, rob;
cout << " Input array a[6]:" << endl;
for (i = 0; i < n; i++)  cin >> a[i];
/*  цикл кроків сортування
(i – номер кроку) */
for (i = 1; i < n; i++)
{
        rob = a[i];
//цикл пошуку місця елементу-вставки в
//масиві

for (j = i; j > 0 && rob < a[j - 1]; j--)
        a[j] = a[j - 1];

// зсув елементів на одну позицію вправо
a[j] = rob;       // вставка елементу
}
// виведення відсортованого масиву
cout << endl;
cout << "\n Sorted array:" << endl;
for (i = 0; i < n; i++)
        cout << a[i] << "   ";
return 0;
}
```

Результати виконання програми:

```
Input array a[6]:
23 -45 39 44 71 12 -60


 Sorted array:
-45   12   23   39   44   71
```

Рисунок 6.8 – Схема алгоритму та програма прикладу 6.14 (метод вставки)

Наведемо ще один варіант програмної реалізації сортування масиву за спаданням елементів з використанням методу вставки, в якій використано покажчики та застосовано цикл *while* для пошуку місця елемента-вставки.

```
#include<iostream>

using namespace std;
int main()
{
        const int n = 6;
        int a[n], i, j, rob;
        int* pmas;
        pmas = a;   // pmas= &a[0]; – покажчик на початок масиву
        cout << "***** Input 6 elements of array:" << endl;
        for (i = 0; i < n; i++)  cin >> *pmas++;
        pmas = a;    // pmas-=n; – повертаємо покажчик на початок масиву
        for (i = 1; i < n; i++)
        {
                rob = *(pmas + i);   j = i;
                while (j > 0 && rob > *(pmas + j - 1))
                {
                        *(pmas + j) = *(pmas + j - 1);  j--;
                }
                *(pmas + j) = rob;
        }
        cout << "Sorted array" << endl;
        for (i = 0; i < n; i++)  cout << *(pmas + i) << "   ";
return 0;
}
```

Результати виконання програми:

```
***** Input 6 elements of array:
-12 34 85 25 0 49
Sorted array
85   49   34   25   0   -12
```

Приклад 6.15.

З двох впорядкованих за зростанням масивів a_i ***(i = 0...n-1)*** і b_j ***(j = 0...m-1)*** сформувати третій, також впорядкований, масив без додаткового сортування (***n = 8, m = 12***).

Алгоритм **злиття двох відсортованих масивів** (рис. 6.9) починається з порівняння перших елементів відповідних масивів ***a*** і ***b***. За лічильником ***i*** вибиратимемо елементи з масиву ***a***, за лічильником ***j*** — з масиву ***b***, а за параметром ***k*** — заносити елементи до масиву ***c***. У масив ***c*** заноситься менший з елементів, що порівнюються, а далі

в порівнянні бере участь наступний елемент того масиву, елемент якого вже записаний до ***c***. Ця процедура повторюється, доки один з масивів не закінчиться. Наприкінці треба тільки переписати до масиву ***c*** всі елементи іншого масиву, що залишилися.

```
#include<iostream>
using namespace std;

int main()
{
        const int n = 8, m = 12;
        int a[n], b[m], c[n + m];
        int i, j, k, l;
// введення двох відсортованих масивів
        cout << "Input 2 sorted arrays:\nArray a[8]:" << endl;
        for (i = 0; i < n; i++)     cin >> a[i];
        cout << "\nArray b[12]:" << endl;
        for (j = 0; j < m; j++)  cin >> b[j];
// злиття масивів a[] і b[] в масив c[]
        for (i = 0, j = 0, k = 0; i < n && j < m; k++)
                if (a[i] < b[j])  c[k] = a[i++];
                else  c[k] = b[j++];
        if (i == n)
                for (l = k; l < n + m; l++) c[l] = b[j++];
        else
                for (l = k; l < n + m; l++) c[l] = a[i++];
// виведення отриманого масиву c[]
        cout << "Rezult: array c[]:" << endl;
        for (i = 0; i < n + m; i++)     cout << c[i] << "  ";
return 0;
}
```

Результати:

```
Input 2 sorted arrays:
Array a[8]:
3 5 6 12 18 34 68 80

Array b[12]:
-3 -1 0 2 6 8 10 12 23 25 37 99
Rezult: array c[]:
-3  -1  0  2  3  5  6  6  8  10  12  12  18  23  25  34  37  68  80  99
```

Ефективний метод сортування великих масивів — **швидке сортування** — розглянутий у підрозділі «Рекурсивні функції».

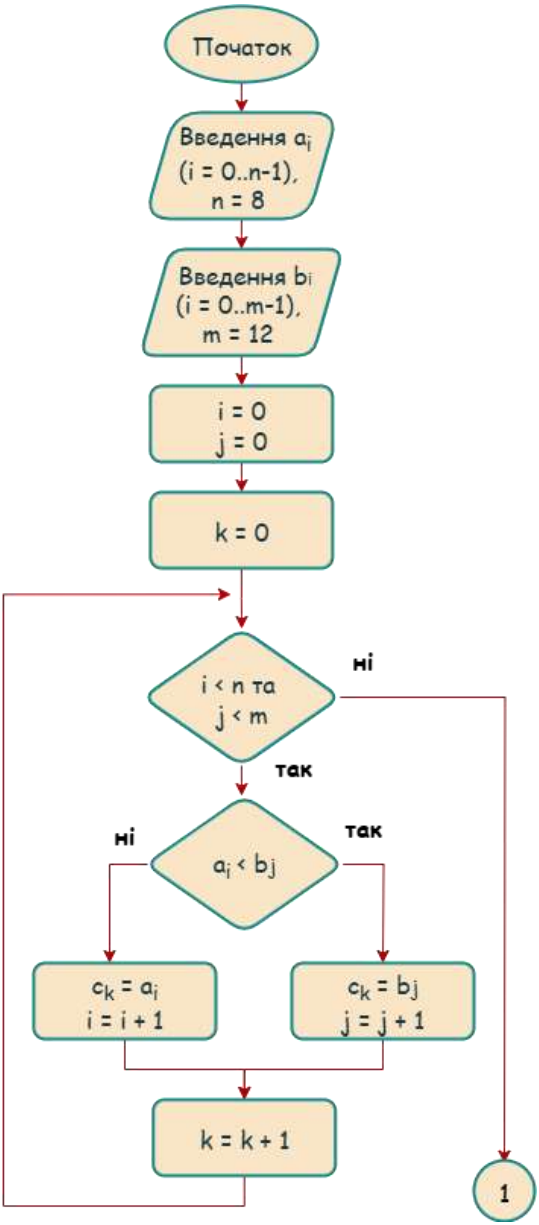


Рисунок 6.9, а – Схема алгоритму ***прикладу 6.15*** (злиття масивів)

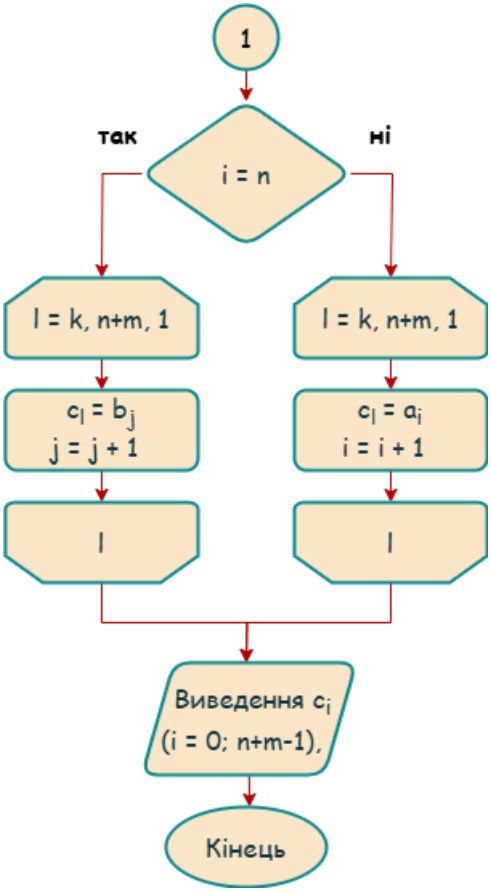


Рисунок 6.9, б – Схема алгоритму ***прикладу 6.15*** (продовження)

6.5 Динамічні масиви

В основній пам'яті дані можуть зберігатися двома способами:

– пам'ять виділяється або в сегменті стека та залишається закріпленою до завершення виконання конкретної функції, або виділяється в сегменті даних на весь час виконання програми;

– пам'ять виділяється в міру потреби (динамічне виділення пам'яті).

Треба зазначити, що всі приклади, розглянуті раніше, демонструють роботу з даними, які зберігаються першим способом.

Динамічна пам'ять — це вільна пам'ять, у якій під час виконання програми можна виділяти місце залежно від потреб користувача. ***Доступ до виділених ділянок динамічної пам'яті, що називаються динамічними змінними, здійснюється тільки через покажчики***. Час існування динамічних змінних — від початку створення до кінця програми або до явного звільнення пам'яті.

У мові С++ застосовують два способи роботи з динамічною пам'яттю. Перший з них дістався в спадщину від мови С і використовує сукупність функцій **malloc()**, другий — працює з операціями **new** та **delete**, які здійснюють динамічний розподіл і звільнення пам'яті з вищим пріоритетом, ніж інші функції.

Оператор **new** виділяє пам'ять і повертає її адресу. За допомогою оператора **delete** відбувається звільнення пам'яті, на яку вказує змінна-покажчик.

Загальна форма запису оператора **new**:

```
змінна-покажчик = new тип змінної;
```

Оператор **delete** має вигляд:

```
delete [ ] змінна-покажчик;
```

Динамічні масиви створюють за допомогою операції **new**, при цьому необхідно вказати їхній тип і розмірність. Наприклад, для одновимірного масиву дійсних чисел, що має 100 елементів, треба записати:

```
     int n = 100; float*p = new float[n];
//змінна-покажчик на float виділяє у динамічній пам'яті ділянку для
//розміщення 100 елементів дійсного типу.
```

Треба пам'ятати, що ***динамічні масиви при створенні не можна ні ініціювати, ні обнулити***.

Приклад 6.16.

З використанням динамічної пам'яті обробити відомість успішності групи студентів з дисципліни «Програмування», підрахувавши середній бал групи та кількість відмінників.

```
#include<iostream>
using namespace std;
int main()
{
        setlocale(LC_ALL, "ukr"); // для використання укр.шрифту
        const int n = 10;    // кількість студентів
        int* mas;            // оголошення покажчика на масив
        int i, k = 0;        // k – кількість відмінників
        double  s = 0;       // s – сума оцінок групи
```

```
        mas = new int[n];   // виділення динамічної пам'яті
        cout << " Введення оцінок: " << endl;
        for (i = 0; i < n; i++)   cin >> mas[i];
        //------ підрахунок суми оцінок і кількості відмінників
        for (i = 0; i < n; i++)
        {
                s = s + mas[i];
                if (mas[i] >= 90)    k++;
        }
        cout << endl;
        cout.precision(3);
        cout << "Середній бал =  " << s / n << endl;
        cout << "Кількість відмінників = " << k << endl;
        delete[]mas;    // звільнення динамічної пам'яті
return 0;
}
```

Результати обчислень:

```
Введення оцінок:
67 89 90 92 100 45 87 74 93 99

Середній бал =  83.6
Кількість відмінників = 5
```

У цій програмі змінна *s* служить для обчислення суми оцінок групи, а змінна ***k*** — для підрахунку кількості відмінників. Перед обчисленням треба надати цим змінним початкове нульове значення (накопичення суми та кількості пояснено у *прикладах 1.1–1.2*).

При створенні динамічного багатовимірного масиву необхідно в операції **new** вказати всі його розмірності (перший вимір може бути змінною), наприклад:

```
    int n = 7; // n – кількість рядків матриці
    int **m = (int **) new int [n][5];
```

Розглянемо більш універсальний і безпечний спосіб виділення динамічної пам'яті для двовимірного масиву, коли обидві його розмірності задаються на етапі виконання програми. Наприклад, розподіл динамічної пам'яті для матриці, що має ***n*** рядків і ***m*** стовпців та елементи цілого типу, можна здійснити так:

```
    int n, m;
    cout<<" Введіть кількість рядків і стовпців : ";
    cin >> n >> m;
    int **a = new int *[n];    /*оголошення змінної типу «покажчик
на покажчик на int» і виділення пам'яті для масиву покажчиків на
рядки матриці*/
    for (int i = 0;  i < n; i++)
```

```
    a[i] = new int [m]; /*кожному елементу масиву покажчиків на рядки присвоюється адреса початку ділянки пам'яті, виділеної для рядка матриці*/
```

Наочно це подано на рисунку 6.10.

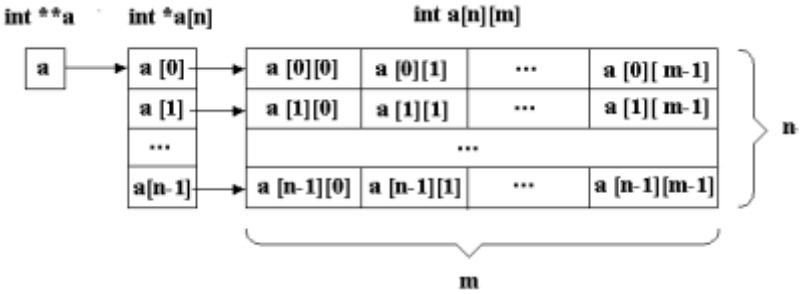


Рисунок 6.10 – Виділення пам'яті під двовимірний масив

Звільнення пам'яті з-під масиву будь-якої кількості вимірів виконується за допомогою операції **delete[ ]**.

Приклад 6.17.

З використанням динамічної пам'яті створити програму обчислення матриці ***C*** за формулою

$$C[n][q] = A[n][m] * B[m][q].$$

Згідно з умовою, елемент матриці ***C[n][q]*** дорівнює сумі добутків елементів відповідного рядка матриці ***A[n][m]*** на елементи відповідного стовпця матриці ***B[m][q]***.

У запропонованій програмі для обчислення елементів добутку матриць організовано три вкладених цикли: цикл перебору рядків матриці ***A***, цикл перебору стовпців матриці ***B***, а також цикл накопичення суми для одержання елемента матриці ***C***.

Для виділення динамічної пам'яті під двовимірні масиви ***A[n][m]***, ***B[m][q]*** і ***C[n][q]*** скористаємося розглянутим вище способом.

```
#include<iostream>
using namespace std;
int main()
{
        int i, j, k;
        int n, m, q;
        cout << " Enter the matrix dimensions: \n";
        cout << " n = "; cin >> n;
        cout << " m = "; cin >> m;
        cout << " q = "; cin >> q;
```

```
// створення динамічних масивів
/*   виділення динамічної пам’яті під масиви покажчиків
                та ініціалізація масивів покажчиків */

        int** C = new int *[n];
        for (int i = 0; i < n; i++)
                C[i] = new int[q];

        int** A = new int *[n];
        for (int i = 0; i < n; i++)
                A[i] = new int[m];

        int** B = new int *[m];
        for (int i = 0; i < m; i++)
                B[i] = new int[q];

//-------------- введення матриць A[n][m] та B[m][q]
        cout << "\n Enter the matrix A[n][m] \n";
        for (i = 0; i < n; i++)
                for (j = 0; j < m; j++) cin >> *(A[i] + j);
        cout << "\n Enter the matrix B[m][q] \n";
        for (i = 0; i < m; i++)
                for (j = 0; j < q; j++) cin >> *(B[i] + j);

// ------------- обчислення матриці C[n][q] = A[n][m] * B[m][q]
        for (i = 0; i < n; i++) // перебір рядків матриці A[n][m]
        for (k = 0; k < q; k++) // перебір стовпців матриці B[m][q]
                {
                        C[i][k] = 0;

        for (j = 0; j < m; j++)
C[i][k] += (A[i][j] * B[j][k]); //визначення елементу C[i][k]
                }
//---------------- виведення на екран матриці C[n][q]
        cout << "\n Matrix C[n][q] = A[n][m] * B[m][q] \n";
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < q; j++) cout << *(C[i] + j) << " ";
                cout << endl;
        }

//---------------- звільнення пам’яті
        for (int i = 0; i < n; i++) delete C[i];
        delete[]C;
        for (int i = 0; i < n; i++) delete A[i];
        delete[] A;
        for (int i = 0; i < m; i++) delete B[i];
        delete[]B;
return 0;
}
```

Результати обчислень:

```
 Enter the matrix dimensions:
 n = 4
 m = 3
 q = 5

Enter the matrix A[n][m]
1 1 1
3 3 3
2 2 2
4 4 4

Enter the matrix B[m][q]
8 8 8 8 8
7 7 7 7 7
5 5 5 5 5

 Matrix C[n][q] = A[n][m] * B[m][q]
20  20  20  20  20
60  60  60  60  60
40  40  40  40  40
80  80  80  80  80
```

6.6 Запитання та завдання

1. Що таке масив та які існують різновиди масивів?
2. Як здійснюється звернення до елементів масивів?
3. Як у C++ реалізується введення-виведення елементів масиву?
4. Охарактеризуйте поняття «покажчик» та наведіть приклади.
5. Які операції дозволені для змінних-покажчиків?
6. Що таке масиви покажчиків та які особливості їхнього використання?
7. Охарактеризуйте алгоритм сортування за методом «бульбашки».
8. Охарактеризуйте алгоритми сортування за методом вибору.
9. Охарактеризуйте поняття «динамічна пам’ять» та її можливості.
10. Як обробляються масиви з використанням динамічної пам’яті?

Завдання.

Набути практичні навички розв’язання задач з одновимірними масивами під час виконання завдань розділу вправ «Одновимірні та двовимірні масиви», а також розділу «Циклічні обчислювальні процеси», якщо вважати, що аргументи обчислювальних функцій задані у вигляді одновимірних масивів.

7 ДАНІ СИМВОЛЬНОГО ТИПУ

Ефективність мови С++ багато в чому визначається наявністю в ній розвинутих засобів для обробки символьної інформації. У стандартній бібліотеці С++ передбачено багато функцій, що виконують прості дії з символьними даними. Тому ця мова найкраще підходить для системної роботи: написання компіляторів, інтерпретаторів, операційних систем, редакторів тексту тощо.

7.1 Рядки як символьні масиви

У мові С та в ранніх версіях мови С++ рядки розглядалися як символьні масиви, та і вся робота з ними ґрунтувалася на використанні цих масивів. Розроблена бібліотека функцій **string.h** містить потужні засоби для роботи з рядковими масивами.

Рядок є масивом символів, який закінчується нуль-символом. Нагадаємо, що нуль-символ має код, що дорівнює 0, і запис у вигляді керуючої послідовності *'\0'*. За розташуванням нуль-символу визначається фактична довжина рядка. Кількість елементів символьного масиву складається з кількості символів у рядку плюс 1, тому що нуль-символ також є елементом масиву.

Для опису рядка використовуються звичайні засоби опису масивів, наприклад: *char str [25];*. Індексування такого масиву, як і будь-якого іншого, починається з нуля.

Символьні послідовності, розділені тільки пробілами, розглядаються як один рядок, тобто запис:

```
"Одна година сьогодні
коштує двох годин завтра."
```

ідентичний до рядку:

```
"Одна година сьогодні коштує двох годин завтра."
```

Адреса першого символу рядка може використовуватися по-різному:

– якщо рядок застосовується при ініціюванні масиву типу *char*, адреса його першого елемента стає синонімом імені масиву. Наприклад, ідентичними є такі описи масиву:

```
char st [ ] = "Слово";
char st [6] = "Слово";
char st [6] = {'С''л''о''в''о''\0'};
```

– якщо рядок використовується для ініціювання покажчика типу *char**, адреса першого символа рядка буде початковим значенням покажчика, наприклад:

```
char *pst = "Слово"; .
```

Тут описується змінна-покажчик *pst*, яка отримує початкове значення, що дорівнює адресі першого елемента (символу *'С'*).

Необхідно звернути увагу на те, що під час опису символьного масиву його ім'я— не змінна, а покажчик-константа на початок рядка, тому її не можна використовувати в деяких операціях адресної арифметики. Зокрема, не можна здійснювати операцію присвоювання вигляду:

```
char st [20];
st = "Петренко"; //неправильно, не можна змінити значення st.
```

Виконання дій з елементами символьного масиву здійснюється через індекси або через покажчики. Для доступу до будь-якого символу рядка використовується індекс масиву *char*. Тобто, якщо описана змінна *char str [3];*, то третім елементом масиву можна скористатися, записавши: *str [2]* або **(str+2)*.

У процесі роботи з елементами двовимірного масиву застосовують або індекси масиву, або індекси покажчиків. Якщо описаний список прізвищ *char spysok[5][15];*, то для використання символу масиву треба записати: *spysok[i][j]* або **(spysok [i] + j).*

Аналогічно, якщо оголошений масив покажчиків *char*str [5]*, що містить 5 елементів, кожний з яких вказує на рядок, то доступ до символу рядка можна здійснити з використанням запису **(str [i] + j)*.

7.2 Введення-виведення символьних масивів

Введення рядків можна здійснювати різними способами, найбільш розповсюдженими з яких є:

– введення шляхом ініціювання при оголошенні символьних масивів:

```
char st [] = "String";
char st1 [7] = "String";
char st2 [7] = {'S','t','r','i','n','g','\0'};
```

У цьому випадку двовимірні масиви можна ініціювати по-різному, наприклад, у вигляді:

```
char str1[5][20] = {"12345", "abcd"};
char str2[][20] = {"12345", "abcd"};
const char* pst1[5] = {"123345", "abcd"};
const char* pst2[] = {"12345", "abcd"};
```

– використання потокового введення cin >>. Здійснюється у випадку, коли рядок не містить пробілів, тому що символ пробілу є роздільником введення даних, наприклад:

```
char  st [5]; cin >> st;
```

– посимвольне введення за допомогою функції *get()*, наприклад:

```
get (st[i]);
```

– введення за допомогою функції *cin.get()*:

```
cin.get (str[i], size, endl);//size – кількість символів, що читаються;
```

– введення з використанням функції *cin.getline()*:

```
cin.getline (str[i], sizeof (str[i]1));
//sizeof() – функція визначення розміру рядка.
```

Виведення рядкових даних реалізується з використанням стандартного вихідного потоку *cout:*

```
cout << st;
cout.write(st, size);
```

тощо.

Для потокового введення/виведення доцільно застосовувати функції *setw(w), setprecision(d), cout.width(w) і cout.precision(d).*

Введення-виведення символьних масивів можна здійснити за допомогою відповідних функцій заголовного файла *stdio.h*, наприклад:

– для введення рядків — *gets(st);* та *scanf (% s, st);*

– для виведення рядків — *puts(st);* і *printf(% s, st)***.**

7.3 Основні функції обробки символьних типів

У ранніх версіях C++ рядки розглядалися як символьні масиви. Для роботи з ними розроблено бібліотеку функцій **string.h**, що містить ефективні засоби для роботи з рядками. Згодом була розроблена стандартна бібліотека шаблонів **Standard Template Library (STL)**, яка надає більш потужні засоби, об'єднані в клас **string**. Але незважаючи на існування цього окремого для рядків класу, символьні масиви, що закінчуються нульовим байтом '\0', залишаються досить популярними. Це відбувається завдяки їхній ефективності та можливості контролювання операції з рядками.

Для обробки символьних типів даних бібліотека функцій **string.h** має велику кількість вбудованих функцій, що збільшують продуктивність праці програмістів та скорочують час на розробку програм, наприклад:

- функції перевірки символів;
- функції перетворення символів;

– функції перевірки рядків;
– функції маніпулювання рядками.

Функції наводяться у вигляді списків, що згруповані за їхнім розташуванням у заголовних файлах. Найчастіше надаються прототипи функцій, що описують, як треба використовувати функції у програмах (розділ 9).

Далі розглянемо прототипи, стислий опис, дію та методику застосування основних функцій обробки даних символьного типу.

Функції копіювання рядків:

– char strcpy (s, *st); — виконує операцію копіювання байтів рядка *st* у рядок *s* (включаючи «\0»; повертає *s*),наприклад:

```
char str[50]="Good day!";
cout<<strcpy(str," Example of strcpy()");// Example of strcpy()
```

– char *strdup (const char *str) — виконує копіювання рядка *str* і повертає покажчик на рядок-копію, наприклад:

```
char st1[] = "we are the champions";
char* st2 = _strdup(st1);
cout << st2; //we are the champions
```

– char * strncpy (char *st1, const char *st2, int n); — виконує копіювання **n** символів з рядка *st2* у *st1* (рядок *st1* має бути більше або дорівнювати *st2*, інакше виникне помилка), наприклад:

```
char st1[] = "1234 5678 90";
char st2[] = "hello";
cout<<strncpy(st1, st2, 3);// hel4 5678 90
```

Функції конкатенації рядків:

– char *strcat (char *st1, const char *st2); — поєднує *st1* і *st2* та повертає *st1*,наприклад:

```
char str[100];
char*str1=strcpy(str, "Visual  ");
str1=strcat(str, "Studio");
cout << str1; //Visual  Studio
```

– char *strncat (char *st1, const char *st2, int n); — додає до рядка *st1 n* символів рядка *st2* і повертає знову в *st1*, наприклад:

```
char st1[25] = "Hello ";
char st2[25] = "students and teachers";
char*str = strncat(st1, st2, 9);
cout << str; //Hello students
```

Функції порівняння рядків:

– int strcmp (char *st1, char *st2); — порівнює рядки *st1* і *st2* та повертає цілу величину, що дорівнює:

< 0 — якщо *st1< st2*;
= 0 — якщо *st1 = st2*;
> 0 — якщо *st1 > st2*; ,
наприклад:

```
char st1[] = "WORD";
char st2[] = "word";
int k = strcmp(st1, st2);
cout << k;                    // -1;
```

– **int stricmp (const char *st1, const char *st2);** — виконує порівняння рядків, не враховуючи регістра символів; повертає цілу величину, як і функція **strcmp()**, наприклад:

```
char st1[] = "WORD";
char st2[] = "word";
int k = _stricmp(st1, st2);
cout << k;                   //0
```

– **int strncmp (char *st1, char *st2, int n);** — виконує порівняння рядків із заданою кількістю символів *n* у *st1* і *st2* і повертає цілу величину:

< 0 — якщо *st1 < st2;*
= 0 — якщо *st1 = st2;*
> 0 — якщо *st1 > st2;*

```
char st1[] = "woRD";
char st2[] = "word";
int k = strncmp(st1, st2,2);
cout << k;                   //0
```

– char *strnicmp (char *st1, char *st2, int n); — виконує порівняння рядків із заданою кількістю символів *n* у *st1* і *st2*, незалежно від регістра, і повертає цілу величину, як і в попередньому випадку.

Функція визначення довжини рядка:

– strlen (st) — повертає довжину змінної *st* без нуль-термінатора *«\0»:*

```
char s[] = " ABCD 1234 abcd/.,*";
cout << strlen(s);          //19
```

Функції перетворення символів рядка:

– char *strlwr (char*st); — перетворює символи рядка *st* верхнього регістра в символи нижнього регістра, при цьому інші символи не враховуються. Наприклад:

```
char st[] = "Programming Language C++";
_strlwr(st);
cout << st;         //programming language c++
```

– char *strupr (char *st); — перетворює символи рядка st нижнього регістра в символи верхнього регістра, інші символи не враховуються;

– char *strrev (char *st); — записує символи в рядку *st* у зворотному порядку (реверсує рядок), наприклад:

```
char st[] = " Hello";
_strrev(st);
cout << st;         //olleH
```

– char *strchr (char *st, int c); — визначає перше входження символу *c* у рядок *st*; повертає покажчик на символ у рядку *st*, що відповідає введеному символу, наприклад:

```
char st[90] = "Programming";
char* spt;
spt = strchr(st, 'r');
cout << spt; //rogramming – тепер покажчик spt вказує на
             //підрядок рядка st
```

– char *strrchr (char *st, int c); — знаходить останнє входження символу *c* у рядок *st;* якщо символ *c* у рядку не виявлений, повертає 0, інакше повертає покажчик на останній символ у рядку *st*, що відповідає заданому зразку, наприклад:

```
char st[90] = "Programming";
char* spt;
spt = strrchr(st, 'r');
cout << spt;                //ramming
```

Функції пошуку підрядка в рядку:

– strspn (const char *st1, const char *st2); — повертає кількість символів від початку рядка *st1,* що збігаються із символами рядка *st2*, де б вони не знаходилися в *st2*, наприклад:

```
char st1[] = "1654-Kharkiv";
char st2[] = "1234567890";
int k = strspn(st1, st2);
cout << k;   //4   - 4 перші символи рядка st1 збігаються з st2
```

– char *strstr (const char *st1, const char *st2); — функція шукає в рядку *st1* перше входження *st2* і повертає покажчик на перший символ, знайдений у *st1*, з підрядка *st2*; якщо рядок *st2* не виявлений в *st1*, функція повертає 0, наприклад:

```
char st1[] = "Better late than never";
char st2[] = "late";
char*p = strstr (st1, st2);
cout << p;          //late than never
```

За потреби визначення останнього входження можна спочатку реверсувати рядок за допомогою функції **strrev().**

– char *strtok (char *st, const char *dlm); — розбиття рядка на лексеми (сегменти), обмежені символами, що входять до складу рядка *dlm*. Цей параметр може містити будь-яку кількість різних обмежників — ознак границь лексем, після виділення лексеми в рядок *st* поміщається символ «\0».

Наступні виклики функції **strtok**() мають бути з першим аргументом *NULL*. Вони повертатимуть покажчик на інші, наявні в *st* лексеми. Щоразу після завершення виділення лексеми у її кінці замість розділового символу поміщається символ «\0». Після того, як у рядку не залишиться жодної лексеми, функція повертає *NULL*. Для збереження вихідного рядка його треба записати в резервну змінну. Цю функцію зручно використовувати для розбиття речення на слова або будь-які інші сегменти. Розглянемо приклад програми з використанням функції **strtok()**.

Приклад 7.1.

Скласти програму, яка вводить речення, здійснює розбиття його на слова, підраховує кількість символів у кожному слові та виводить відповідну інформацію (реалізація в MS Visual Studio).

```
#define _CRT_SECURE_NO_WARNINGS //тільки для MS Visual Studio
#include<iostream>
#include <string>

using namespace std;
int main()
{
        char* tk;
        char spt[] = ", : .!";
        char st[] = "To be, or not to be, that is the question:";
        cout << st << endl;
        int i = 1;
        tk = strtok(st, spt);
        while (tk != NULL)
```

```
        {
     cout <<"word "<< i << " = " << tk;
     cout << " - contains " << strlen(tk) <<" symbols"<< endl;
              tk = strtok(NULL, spt);
      i++;
        }
return 0;
}
```

Результати виконання програми:

```
To be, or not to be, that is the question:
word 1 = To - contains 2 symbols
word 2 = be - contains 2 symbols
word 3 = or - contains 2 symbols
word 4 = not - contains 3 symbols
word 5 = to - contains 2 symbols
word 6 = be - contains 2 symbols
word 7 = that - contains 4 symbols
word 8 = is - contains 2 symbols
word 9 = the - contains 3 symbols
word 10 = question - contains 8 symbols
```

В розглянутому прикладі **#define _CRT_SECURE_NO_WARNINGS** є директивою препроцесора, що використовується в Microsoft Visual Studio для відключення попереджень про небезпеку використання застарілих або небезпечних функцій бібліотеки C, таких як scanf(), gets(), strcpy(), sprintf() тощо.

За замовчуванням, Visual Studio видає попередження при використанні цих функцій, оскільки вони можуть призвести до проблем із безпекою, наприклад, переповнення буфера. Microsoft рекомендує використовувати безпечніші альтернативи, наприклад, scanf_s(), strcpy_s(), і т.д. Директива просто відключає ці попередження, дозволяючи використовувати традиційні, менш безпечні функції без зауважень компілятора.

Процес розбиття речення на слова можна було б здійснити з використанням і такого програмного фрагмента:

```
     tk = strtok (st< spt);      // перший виклик функції
     while (tk)
     if ((tk = strtok(), spt) != 0) cout <<....
```

Для видалення з рядка підрядка *або* ***символу із заданої позиції*** у бібліотеці **string.h** немає спеціальної функції, однак можна написати власну, наприклад:

```
void del(char* st, int k, int n);
{
for (int i = k; i < strlen(st); i++)
st[i] = st[i + n];
st[i] = '\0';       // запис '\0' в кінець нового рядка
}
```

де *st* — вихідний рядок (покажчик на нього);

n — кількість символів у підрядку, що вилучається;

k — позиція, з якої треба вилучити підрядок.

Наведемо приклад, який ілюструє використання функції **void del ();** (детальніше про функції дивись у розділі 9).

Приклад 7.2.

Скласти програму вилучення підрядка в ***n*** символів з ***k***-ї позиції в рядку.

```
#define _CRT_SECURE_NO_WARNINGS //тільки для MS Visual Studio
#include<iostream>
#include <string>
#include <windows.h> //для SetConsole()

using namespace std;
   //--------------- функція видалення підрядка з рядка
   void del(char* sp, int k, int n)
   {
           int i;
           for (i = k; i < strlen(sp); i++)
                   sp[i] = sp[i + n];
           sp[i] = '\0';
   }
int main() //---- головна функція
{
SetConsoleOutputCP(1251);
SetConsoleCP(1251); //для українського шрифту

   char st[50], pst[10];
   cout << "***** Введіть рядок\n";
   cin.getline(st, 50);
   cout << "***** Введіть підрядок\n";
   cin >> pst;
   cout << "Початковий рядок: –  " << st << endl;
           if (strstr(st, pst) != NULL)
                   del(st, strstr(st, pst) - st, strlen(pst));
           cout << "Новий рядок: –  " << st << endl;
return 0;
}
```

Результати:

```
***** Введіть рядок
Люблю писати програми мовою С++!
***** Введіть підрядок
писати
Початковий рядок: – Люблю писати програми мовою С++!
Новий рядок: –  Люблю програми мовою С++!
```

– **void* memchr (const void *st, int s, int n);** — функція шукає символ «*s*» у рядку **st* довжиною *n* байт, тобто в блоці пам'яті, на який вказує покажчик *st*. Якщо символ *s* знайдений, функція повертає покажчик на цей символ, а в протилежному випадку — повертає *NULL*:

```
     char str[] = "Honesty is the best policy";
     char ch = 'i';
cout << "String after memchr(): ";
cout << (char*)memchr(str, ch, strlen(str));
     //String after memchr(): is the best policy
```

– void* memcmp (const void *s1, const void *s2, n); і **void* memicmp (const void *s1, const void *s2, int count);** — функції порівнюють *n* байт з двох буферів, на початок яких указують *s1* і *s2*.

Функції повертають значення

< 0 — якщо *s1 < s2;*

= 0 — якщо *s1 = s2;*

> 0 — якщо *s1 > s2;*

– char *strset (char *st, int ch, int n); і **char *strset (char *st, int ch);** — функції заповнюють рядок *st* символом *ch* і повертають покажчик на отриманий рядок, *n* — заповнює *n* символів рядка *st*:

```
     char str[] = "Original String\n";
     cout << str;         //Original String
     cout << "String after strnset(): " << _strnset(str, '*', 8);
     //String after strnset(): ******** String
```

Функції перетворення рядків у числа та чисел у рядки знаходяться у файлі **stdlib.h**:

– int atoi (const char *s); — перетворює рядок *s* у число типу *int*. Повертає отримане число 0, якщо зустрінеться символ, що не може бути перетворений. Рядок повинен містити число, наприклад, «2345», та мати таку структуру:

[пробіли] [знак числа] [цифри];

```
     int i, i1;
     const char *s = "-12345";
     const char* s1 = "12abc";
```

```
i = atoi(s);
cout << i;                    //-1234 як ціле число
cout<< (i1 = atoi(s1));       //12
```

– long atol (const char *s); — перетворює рядок *s* у число типу *long int* (аналогічно функції **atoi()**);

– double atof (const char *s); — перетворює рядок символів у число з плаваючою крапкою типу *double*. Якщо зустрічається символ, що не може бути перетворений, повертає 0. Оброблюваний рядок повинен мати таку структуру:

[пробіли] [знак числа] [цифра.цифра] [літера e, E, d або D] [знак порядку] [цифри порядку],

наприклад, «12345.123» або «12.345123 E3»;

```
double x;
char s[] = " -10203.12E-15";
x = atof(s);
cout << x;          //-1.02031e-11 як дійсне число
```

– char *ecvt (double vl, int n, int *dec, int *sign); — перетворює число *vl* у рядок символів, кількість яких дорівнює *n* символів цифр. Положення десяткової крапки від першої цифри числа повертається до змінної, на яку вказує *dec*. Знак числа повертається до змінної, на яку вказує *sign*. Якщо *sign = 0*, то число є додатним, інакше — від'ємним. Отриманий рядок зберігається у внутрішній пам'яті функції, покажчик повертається на початок сформованого рядка:

```
int  dec, sign;
char* str;
double value = 3.1415926535;
cout << "\nvalue = " << value;
str = _ecvt(value, 5, &dec, &sign);
cout << "\n n=5 string = "<< str;
cout << "  dec = " << dec << "  sign = "<<sign;
//n=5 string = 31416  dec = 1  sign = 0
str = _ecvt(value, 10, &dec, &sign);
cout << "\n n=10 string = " << str;
cout << "  dec = " << dec << "  sign = " << sign;
// n=10 string = 3141592654  dec = 1  sign = 0
```

– char *fcvt (double vl, int n, int *dec, int *sign); — аналогічна до попередньої функції **char *ecvt()**, але якщо для функції *ecvt* параметр *dec* задає загальну кількість цифр, то для функції ***fcvt*** — кількість цифр після десяткової крапки;

– char *gcvt (double vl, int n, char *buf); — перетворює число *vl* у рядок, котрий поміщає в буфер, покажчик на початок якого є

buf, n — число цифр у символічному записі перетвореного числа. Отриманий рядок містить символ знаку числа і десяткової крапки, якщо число містить менше десяткових цифр, ніж *n*. У цьому випадку молодша цифра дробової частини відкидається. Якщо перетворене число не можна помістити в задану кількість цифр *n*, функція генерує символьний запис в експоненціальній формі із символом *E* і знаком по рядку. Функція повертає покажчик на початок сформованого рядка.

Функції перевірки символів знаходяться у файлі **ctype.h**:

– isgraph (s) — повертає значення «істина» (1), якщо *s* є друкованим символом, і «неправда» (0), якщо *s* є пробілом або яким-небудь не відображуваним символом;

– isprint (s) — повертає значення «істина» (1), якщо *s* є друкованим символом, включаючи символ пробілу, і «неправда» (0) у всіх інших випадках;

– ispunct (s) — повертає значення «істина» (1), якщо *s* є знаком пунктуації (будь-який друкований символ, крім пробілу), і «неправда» (0) в інших випадках;

– isdigit (s) — повертає значення «істина» (1), якщо *s* є цифрою від 0 до 9, і «неправда» (0) в інших випадках;

– isalnum (s) — повертає значення «істина» (1) якщо *s* є цифрою або літерою (заголовною або строковою), і «неправда» (0) у всіх інших випадках (тобто перевіряє алфавітні та цифрові символи).

Функції перетворення символів:

– tolower (s) — перетворює символ *s* до нижнього регістра;
– toupper (s) — перетворює символ *s* до верхнього регістра;

```
char s = 'd';
char digit = '8';
char space = ' ';
char symb = '?';
cout << s << " digit or alfa?: ";
     isalnum(s) ? cout<< " true\n" : cout << "false\n";
     //d digit or alfa?:  true
cout << s << " is alfa?: ";
     isalpha(s) ? cout << " true\n" : cout << "false\n";
     //d is alfa?:  true
cout <<digit <<" is digit?: ";
     isdigit(digit) ? cout <<" true\n" : cout<< "false\n";
     //8 is digit ? : true
cout << space<< " is space?: ";
     isspace(space) ? cout <<" true\n" : cout << "false\n";
```

```
     //  is space?:  true
cout<<symb<<" digit or alfa?: ";
     isalnum(symb) ? cout<<" true\n" : cout << "false\n";
     //? digit or alfa?: false
cout<<"s='d' in upper: "<< char(toupper(s));
     //s='d' in upper: D
```

Розглянемо приклади з використанням функцій обробки рядків.

Приклад 7.3.

Ввести до пам'яті комп'ютера список прізвищ, які розташовані в будь-якому порядку, та відсортувати їх за алфавітом.

Розглянемо перший варіант реалізації поставленої задачі. Вважатимемо, що вводять прізвища та ініціали, тоді програма може мати вигляд:

```
#define _CRT_SECURE_NO_WARNINGS //тільки для MS Visual Studio
#include<iostream>
#include <string>
#include <windows.h> //для SetConsole()
using namespace std;

int main()
{SetConsoleOutputCP(1251);
SetConsoleCP(1251); //для українського шрифту
        const int n = 5;
        char sp[n][20], r[20];
        int i, k;

     // введення прізвищ та ініціалів
        cout << "***** Введіть " << n << " прізвищ  \n";
        for (i = 0; i < n; i++)
        {
        cout << "Введіть " << (i + 1) << " прізвище та ініціали\n";
        cin.getline(sp[i], sizeof(sp[i]) - 1);
        }

     // сортування списку прізвищ
        for (k = 1; k < n; k++)
                for (i = 0; i < n - k; i++)
                        if (strcmp(sp[i], sp[i + 1]) > 0)
                        {       strcpy(r, sp[i]);
                                strcpy(sp[i], sp[i + 1]);
                                strcpy(sp[i + 1], r);
                        }
cout << "\nВідсортований масив прізвищ \n";
        for (i = 0; i < n; i++)
                cout << sp[i] << endl;
return 0;
}
```

Результати:

```
****** Введіть 5 прізвищ
Введіть 1 прізвище та ініціали
Шевченко Т.Г.
Введіть 2 прізвище та ініціали
Костенко Л.В.
Введіть 3 прізвище та ініціали
Андрухович Ю.І.
Введіть 4 прізвище та ініціали
Забужко О.С.
Введіть 5 прізвище та ініціали
Цілик І.А.

Відсортований масив прізвищ
Андрухович Ю.І.
Забужко О.С.
Костенко Л.В.
Цілик І.А.
Шевченко Т.Г.
```

У наведеній програмі використано масив прізвищ ***sp[5][20]*** і символьний рядок ***r***, який потрібен для тимчасового зберігання прізвища при сортуванні масиву. Для сортування був застосований раніше розглянутий метод виштовхування («бульбашки») (підрозділ 6.4).

Порівняння елементів символьного масиву (***char sp[n][20]***) здійснюється за допомогою функції *strcmp()*, а перезапис прізвищ з одного елемента масиву ***sp[i]*** в другий — ***sp[i+1]*** — за допомогою функції *strcpy( )* і змінної ***r***.

Після сортування на екран виведено одержаний масив.

Другий варіант розв'язання поставленої задачі використовує покажчики.

```
#define _CRT_SECURE_NO_WARNINGS //тільки для MS Visual Studio
#include<iostream>
#include <string>
#include <windows.h> //для SetConsole()

using namespace std;

int main()
{
SetConsoleOutputCP(1251);
SetConsoleCP(1251); //для українського шрифту

                const int n = 5;
                char sp[n][20];
                int i, k;
                char* ps[n], * ptr;       // ps[n] – масив покажчиків
```

```
//--------------введення прізвищ, ініціалізація масиву покажчиків
                cout << "***** Введіть прізвища \n";
                for (i = 0; i < n; i++)
                {
                        fgets(sp[i], 20, stdin);
                        ps[i] = sp[i];
                }
//--------------виведення початкової інформації
                cout << "\n***** Початковий список\n";
                for (i = 0; i < n; i++)
                        puts(ps[i]);
//--------------сортування масиву
                for (k = 1; k < n; k++)
                        for (i = 0; i < (n - k); i++)
                                if (strcmp(ps[i], ps[i + 1]) > 0)
                                {
                                        ptr = ps[i];
                                        ps[i] = ps[i + 1];
                                        ps[i + 1] = ptr;
                                }
// ----------- виведення відсортованого масиву
                        cout << "\n\n**** *Відсортований список \n";
                        for (i = 0; i < n; i++)
                                puts(ps[i]);
return 0;
}
```

Результати виконання програми:

```
***** Введіть прізвища
Петренко Ф.І.
Корженко А.Л.
Белобровко А.М.
Апанасенко К.В.
Голобородько В.І.

***** Початковий список
Петренко Ф.І.

Корженко А.Л.

Белобровко А.М.

Апанасенко К.В.

Голобородько В.І.



**** *Відсортований список
Апанасенко К.В.
```

```
Белобровко А.М.

Голобородько В.І.

Корженко А.Л.

Петренко Ф.І.
```

Приклад 7.4.

Ввести рядок і видалити в ньому зайві пробіли.

```
#define _CRT_SECURE_NO_WARNINGS //тільки для MS Visual Studio
#include<iostream>
#include <string>
using namespace std;

int main()
{       char st[] = "  Errare    humanum     est";
        int i, n = 0;       //   n –  для підрахунку пробілів
        for (i = 0; i < strlen(st); i++)
                {
                if (st[i] != ' ')
                        {
                        cout << st[i];
                        n = 0;
                        }
                        else n++;
                                if (n == 1) cout << st[i];
                        }

return 0;
}
```

Результат виконання програми:

```
Errare humanum est
```

Приклад 7.5.

Визначити позицію входження підрядка в рядок.

```
#define _CRT_SECURE_NO_WARNINGS //тільки для MS Visual Studio
#include<iostream>
#include <string>
#include <windows.h> //для SetConsole()

using namespace std;
const int m = 50;
int main()
{
SetConsoleOutputCP(1251);
```

```
SetConsoleCP(1251); //для українського шрифту

        char* pt, mainstr[m], substr[m];
        int n, k = 0;

        cout << "***** Введіть рядок " << endl;
        cin.getline(mainstr, m);
        cout << "***** Введіть підрядок " << endl;
        cin.getline(substr, m);
        pt = strstr(mainstr, substr);
        cout << endl;
        while (pt)

        {
        k++;   // номер входження
        n = pt - mainstr;
        cout << k << " -е входження підрядка";
        cout << "   номер позиції = " << n << endl;
                pt = strstr(++pt, substr);
                // cout << k << "  " << *pt << endl;
                        }
        if (k == 0) cout << "Підрядок не міститься в рядку" << endl;
return 0;
}
```

Результат виконання:

```
***** Введіть рядок
Майже у всіх справах найважче – початок.
***** Введіть підрядок
ай

1 -е входження підрядка   номер позиції = 1
2 -е входження підрядка   номер позиції = 22
```

Приклад 7.6.

Знайти заданий символ у рядку.

```
#define _CRT_SECURE_NO_WARNINGS //тільки для MS Visual Studio
#include<iostream>
#include <string>
#include <windows.h> //для SetConsole()

using namespace std;
const int m = 100;
int main()
{
SetConsoleOutputCP(1251);
SetConsoleCP(1251); //для українського шрифту
        char sim, * pt, str[m];
        int n, k = 0;
```

```
        cout << "***** Введіть рядок" << endl;
        cin.get(str, m);
        cout << "***** Введіть символ" << endl;
        cin >> sim;
        pt = strchr(str, sim);
        while (pt)
        {       k++;
                n = pt - str;
        cout << k << "-а позиція входження символу = " << n << endl;
                pt = strchr(++pt, sim);
                }

        cout << "Кількість входжень = " << k << endl;
        if (k == 0) cout << "Символ не входить в рядок" << endl;
return 0;
}
```

Результат:

```
***** Введіть рядок
Мова програмування – це система позначень для опису алгоритмів і
структур даних
***** Введіть символ
о
1-а позиція входження символу = 1
2-а позиція входження символу = 7
3-а позиція входження символу = 33
4-а позиція входження символу = 46
5-а позиція входження символу = 55
Кількість входжень = 5
```

Приклад 7.7.

З уведеного списку прізвищ (без ініціалів) вилучити такі, що починаються на задану літеру та мають задане закінчення, вивести повідомлення про прізвище з найменшою кількістю літер.

```
#define _CRT_SECURE_NO_WARNINGS //тільки для MS Visual Studio
#include<iostream>
#include <string>
#include <windows.h> //для SetConsole()

using namespace std;
int main()
{
SetConsoleOutputCP(1251);
SetConsoleCP(1251); //для українського шрифту
        const int n = 6;
        char spysok[n][15], pok[5], p;
        int i, minprizv, k = 0;
        //введення списку прізвищ без ініціалів
        cout << "***** Введіть " << n << " прізвищ\n";
```

```
                for (i = 0; i < n; i++)
                cin >> spysok[i];
        cout << "***** Введіть першу букву\n";
        cin >> p;
        cout << "**** Введіть закінчення\n";
        cin >> pok;
        //визначення прізвища на задану букву та на задане закінчення
        cout << "***** Шукані прізвища\n";
                for (i = 0; i < n; i++)
                        if (spysok[i][0] == p &&
                        strcmp(strrchr(spysok[i], pok[0]), pok) == 0)
                cout << spysok[i] << endl;
        //пошук прізвища з найменшою кількістю букв
        minprizv = strlen(spysok[0]);
                for (i = 1; i < n; i++)
                        if (strlen(spysok[i]) < minprizv)
                        {
                        minprizv = strlen(spysok[i]);
                        k = i;
                        }
cout << "\nПрізвище з найменшою кількістю букв – " << spysok[k];
cout << "\nЙого довжина = " << strlen(spysok[k]) << " символів";
return 0;
}
```

Результати обчислень:

```
***** Введіть 6 прізвищ
Андрієнко
Коваленко
Степаненко
Курко
Коноваленко
Перекотиполе
***** Введіть першу букву
К
**** Введіть закінчення
ко
***** Шукані прізвища
Коваленко
Курко
Коноваленко

Прізвище з найменшою кількістю букв – Курко
Його довжина = 5 символів
```

У програмі для зберігання першої літери використовується змінна з ім'ям ***p***, а для закінчення — змінна ***pok***. Вилучення закінчення прізвища здійснює функція ***strrchr(spysok[i], pok[0])***, яка повертає покажчик на останнє входження заданої літери (***pok[0]***) у рядок (***spysok[i]***). Потім цей покажчик порівнюється з введеним закінченням

прізвища за допомогою функції *strcmp()*. У цілому в операторі *if ...* визначається як перша літера, так і закінчення прізвища. Змінна ***minprizv*** існує для знаходження прізвища з найменшою кількістю літер згідно з алгоритмом визначення мінімального елемента масиву (приклад 1.3).

7.4 Використання рядків типу string

Раніше розглядалися питання обробки символьних даних мови С++, у тому числі і символьних рядків. Однак в останніх версіях мови С++, починаючи з С++4.5, введена стандартна бібліотека шаблонів — **Standard Template Library** (**STL**), яка містить клас **string** з більш потужними засобами обробки рядків.

Для підключення цього класу до програми необхідно записати директиву:

#include <string> (без розширення **.h**)

і підключити простір імен бібліотеки шаблонів у вигляді

using namespace std; .

Після цього можна оголошувати змінні типу **string**:

```
string str1, str2;
```

Ініціювання рядків при оголошенні виконується одним із способів:

```
string st1 = "Це рядок класу string";
string st2 ("Це другий рядок класу string");
```

Оголошення покажчика на рядок здійснюється так:

```
string *pst;
```

Пам’ять для покажчика може бути виділена з будь-яким початковим значенням за допомогою функції **new**, наприклад:

```
string *pstr1 = new string; //оголошується порожній рядок
string *pstr2 = new string ("Новий рядок");
//покажчик вказує на рядок «Новий рядок»
```

Раніше оголошеному покажчику **pst*, який ні на що не вказує, можна присвоїти значення у вигляді *pst = new string ("Це перший рядок");* .

Значення рядка **string** містить будь-який набір символів, записаний у лапках.

Для рядків типу **string** визначено такі операції:

– конкатенації (приєднання), котрі позначаються символом «**+**»;

– відношення («**==**»,«**!=**»,«**>**»,«**>=**»,«»,«=»).

Наприклад, фрагмент

```
string st1 = "Dynamic";
string st2 = "memory";
string st3 = st1 + ’ ‘ + st2;
cout << st3 << endl;
```

дозволить вивести на екран повідомлення: *Dynamic memory*.

Для введення рядків **string**, крім операторів присвоювання, застосовують оператори введення даних:

```
cin >> st;
cin.getline(st,sizeof(st)); getline(cin,st, ‘\n’);
```

тощо.

Виведення рядків на екран здійснюється шляхом використання звичайних операторів виведення даних.

Рядки можна об’єднувати у масиви, які оголошуються звичайним засобом, тобто

```
string sp[10]; //оголошення масиву, що містить 10 рядків.
```

Доступ до символів рядка здійснюється шляхом запису порядкового номера символу — індексу, який починається з нуля. Індекси можна записувати як у квадратних дужках, так і у звичайних, круглих.

Наприклад, якщо записати

```
string str = "My string"; ,
```

то *str[3]* — це буде літера *‘s’*.

Для масивів рядків потрібний символ визначається шляхом запису двох індексів: індексу елемента масиву та індексу символу в цьому елементі, тобто у вигляді *mas[i][j].*

Існує багато функцій для обробки рядків типу **string**, розглянемо деякі з них.

Функції визначення довжини рядка:

– str.size();
– str.length();
– **str.max_size();** .

Наприклад:

```
#include<iostream>
#include <string>
using namespace std;
int main()
{
setlocale(LC_ALL, "ukr");
string str, st ="Життя складне і єдиний у ньому підручник – досвід";
str = "єдиний";
```

```
cout <<"Довжина рядка str = "<<str.size();
cout <<" Довжина рядка st = " << st.length();
return 0;
}
```

Результат виконання:

```
Довжина рядка str = 6 Довжина рядка st = 49
```

Функції додавання одного рядка або його частини до іншого рядка:

– str.append(st); — додає рядок *st* до кінця рядка *srt;*

– str.append(st,k,n); — додає до рядка *str n* символів рядка *st*, починаючи з позиції *k*. Наприклад:

```
string str, st = "Substring in text";
str.append(st, 3, 6);
cout<<"str = "<<str; //str = string
```

Функція включення рядка в рядок:

– str.insert(k,st) — вставляє в рядок *str* з позиції *k* рядок *st*;

– str.insert(k1,st,k2,n) — вставляє в рядок *str* з позиції *k1 n* символів рядка *st*, починаючи з позиції *k2* в рядку *st*. Наприклад:

```
string str = "My  text", st = "string ";
str.insert(3, st);
cout << "str = " << str << endl; //str = My string  text
str.insert(3, st, 2, 4);
cout << "str = " << str << endl; //str = My ringstring  text
```

Функція вилучення символів із рядка:

– str.erase (k,n) —вилучає *n* символів з рядка *str*, починаючи з позиції *k*. Наприклад:

```
string str = "Repetitio est mater studiorum";
str.erase(2,9);
cout << "str = " << str; //str = Rest mater studiorum
```

Функція заміни частини рядка або усього рядка:

– str.replace(st) —заміняє рядок *str* на *st*;

– str.replace(k,n,st) —заміняє в рядку *str n* символів, починаючи з позиції *k* рядка *st*;

– str.replace(k1,n1,st,k2,n2) — заміняє в рядку *str n1* символів з позиції *k1* частиною в *n2* символи рядка *st*, починаючи з позиції *k2*.

Наприклад:

```
string str, st = "substring";
str = "My text string";
str.replace(8, 9, st);
cout <<"str = " <<str;
```

```
string str, st = " substring";
str = "My text long";
str.replace(3, 5, st, 1, 3);
cout <<" str = "<< str;
```

Результат:

```
str = My text substring
```

```
str = My sublong
```

Функція обміну змістом двох рядків:

– str.swap(st) — обмінює зміст рядків *str* та *st*. Наприклад:

```
string W = "White", B = "Black";
W.swap(B);
cout << "W = " << W << "  B = " << B;    //W = Black  B = White
```

Функція виділення частини рядка:

– str.substr(k,n) — повертає частину рядка *str* в *n* символів, починаючи з позиції *k*. Наприклад:

```
string str, st;
str = "My string text";
st = str.substr(3, 13);
cout << " st = " << st;                    // st = string text
```

Функція пошуку позиції входження підрядка в рядок:

– str.find(st,k) — шукає зліва граничну позицію входження рядка *st* **у** рядок *str*, починаючи з *k*-ї позиції рядка *str*;

– str.rfind(st,k) — шукає зправа граничну позицію входження рядка *st* в рядок *str*, починаючи з *k*-ї позиції рядка *str*.

Наприклад:

```
string str = "Citius, altius, fortius", st = "tius";
cout << str.find(st, 0)<<endl;                    //2
cout << str.rfind(st, str.length());              //19
```

Функція присвоювання усього рядка або його частини іншому рядку:

– str.assign(st) —присвоює весь рядок *st* типу *string* або масив *char[ ]* — рядку *str* типу *string*;

– str.assign(st,k,n) —присвоює *n* символів рядка *st* рядку *str*, починаючи з *k*-ої позиції.

Ці функції можна використовувати для перетворення рядка типу *char* у рядок *string*. Наприклад:

```
char st[] = "Шануй мудрість, а не золото";
string str;
cout << "str = " << str.assign(st, 5, 9)<<endl;  //str = мудрість
cout << "str = " << str.assign(st, 0, 14);       //str = Шануй мудрість
```

Функція порівняння рядків або їхніх частин:

- str.compare(st) — порівнює рядки *st* та *str* і повертає значення:
< 0 — якщо *st< str;*
= 0 — якщо *st = str;*
> 0 — якщо st > str;

– str.compare(k,n,st) — порівнює *n* символів рядка *st* з рядком *str,* починаючи з *k*-ї позиції. Наприклад:

```
string str = "0123456789", st = "123abcd0";
cout << str.compare(5, 2, st);  //1
cout << str.compare(5, 10, st); //1
```

***Функція перетворення рядка типу* string *у рядок типу* char**:

– str.c_str() — перетворює рядок типу *string* у рядок типу *char*:

```
string str = "This is my string";
char* ch = new char[sizeof(str)];
strcpy(ch, str.c_str()); // копіювання рядка в масив символів
cout << "Original String: " << str << endl;
                                  //Original String: This is my string
cout << "String in array: " << ch; //String in array: This is my string
delete (ch);
```

– str.empty() — функція перевіряє, порожній рядок чи ні, повертає логічне значення *true* (якщо рядок пустий) або *false* (якщо рядок не пустий):

```
string str;
if (str.empty()) cout << "String is empty";
else cout << "String is not empty";
```

Результат:

```
String is empty
```

Розглянемо приклади використання наведених функцій під час обробки рядків типу **string**.

Приклад 7.8.

Увести до пам’яті комп’ютера будь-який текст, відокремити в ньому всі слова, вивести їх на екран та визначити найдовше слово.

```
#include<iostream>
#include<string>

using namespace std;

int main() {
  string text;
     cout << "Input text: \n";
         getline(cin, text);

int start = 0;      //змінні для зберігання початку
int end = 0;        //та кінця поточного слова
int k = 0;          //кількість слів
string longestWord;
int maxLength = 0;  //розмір найдовшого слова

cout << "Words in text:" << endl;
      //-----------------Прохід по всіх символах у рядку
         for (int i = 0; i <= text.length(); ++i) {
      /*якщо досягнутий кінець рядка або поточний символ не буква -
знайдений кінець слова*/
             if (i == text.length() || !isalpha(text[i])) {
      //-------------Якщо було знайдене слово
                 if (end > start) {
                     ++k;
     //---------------Отримуємо слово, використовуючи підрядок
                     string word = text.substr(start, end - start);
                     cout << "word "<<k<<" - "<<word << endl;
     //--------------- Перевіряємо, чи є поточне слово найдовшим
                     if (end - start > maxLength) {
                         maxLength = end - start;
                         longestWord = word;
                     }
                 }
                 start = i + 1;// встановлюємо початок нового слова
                 end = start;
             }
             else {
                 ++end;//збільшуємо індекс кінця поточного слова
             }
         }
         if (!longestWord.empty()) {//якщо було знайдено найдовше слово
             cout << "Longest word: " << longestWord << endl;
             cout << "Size of longest word: " << maxLength;
         }
         else {
             cout << "The text is empty or contains no words";
         }
return 0;
}
```

Результати:

```
Input text:
A journey of a thousand miles begins with a single step
Words in text:
word 1 - A
word 2 - journey
word 3 - of
word 4 - a
word 5 - thousand
word 6 - miles
word 7 - begins
word 8 - with
word 9 - a
word 10 - single
word 11 - step
Longest word: thousand
Size of longest word: 8
```

У програмі текст вводиться за допомогою функції *getline()* (зверніть увагу на її вигляд), а коментарі пояснюють кожен етап роботи.

Приклад 7.9.

Увести список прізвищ і відсортувати його за алфавітом.

```
#include<iostream>
#include<string>

using namespace std;

int main()
{
        const int n = 5;
        string list[n];
        int i, k;
//------------------введення списку прізвищ
        for (i = 0; i < n; i++)
        {
                cout << "***** Enter " << (i + 1) << " name\n";
                getline(cin, list[i], '\n');
        }

//------------------сортування списку прізвищ
        for (k = 1; k < n; k++)
                for (i = 0; i < n - k; i++)
                if (list[i] > list[i + 1])  list[i].swap(list[i + 1]);

//------------------ виведення відсортованого списку
        cout << "\n***** Outputting a sorted list\n";
        for (i = 0; i < n; i++)
```

```
                cout << (i + 1) << "  " << list[i] << endl;
return 0;
}
```

Результати виконання:

```
***** Enter 1 name
Petrenko S.S.
***** Enter 2 name
Menenko K.O.
***** Enter 3 name
Savchuk H.C.
***** Enter 4 name
Petrenko A.N.
***** Enter 5 name
Ageev M.E.

***** Outputting a sorted list
1  Ageev M.E.
2  Menenko K.O.
3  Petrenko A.N.
4  Petrenko S.S.
5  Savchuk H.C.
```

Для порівняння прізвищ у програмі використовується звичайна операція «**>**», а для взаємозаміни — функція **swap()** (проаналізуйте програми прикладу 7.10 і прикладу 7.4).

7.5 Запитання та завдання

1. Що таке рядки та значення елементів символьного типу?
2. Що таке масив символьного типу?
3. Як здійснюється введення символьних даних?
4. Як виконується порівняння даних символьного типу?
5. Наведіть приклад використання операції конкатенації.
6. Як визначити кількість символів у рядку?
7. Які функції мови C++ необхідні для виділення підрядка з рядка?
8. Які функції здійснюють перевірку символів?
9. Як виконується перетворення рядків у числа і навпаки?
10. Охарактеризуйте функції пошуку підрядка в рядку.
11. Які функції здійснюють перевірку символів?
12. Як можна здійснити видалення підрядка з рядка або символу із заданої позиції?

13. Які інструкції треба записати, щоб підключити бібліотеку шаблонів STL до програми?
14. Які операції можна здійснити з рядками типу *string*?
15. Як можна ініціювати рядки типу *string*?
16. Як можна визначити символ у рядках типу *string*?
17. Як здійснюється ініціювання покажчика на рядок типу *string*?

Завдання.

Набуття практичних навичок програмної реалізації задач з даними символьного типу при розв'язанні завдань розділу вправ «Рядки».

8 ДАНІ ТИПУ СТРУКТУРА

8.1 Використання структур

Структура — це сукупність різнотипних елементів, яким присвоюється одне ім'я (воно може бути відсутнім), що займає одну ділянку пам'яті. Елементи, що складають структуру, називаються *полями* [1, 2, 4 — 8, 12 — 14, 18, 19].

Змінна типу **структура**, як і будь-яка змінна, має бути описана. Цей опис складається з двох кроків: опису шаблону (тобто складу) або типу структури та опису змінних структур ного типу.

Синтаксис опису структури має вигляд:

```
struct  [<ім'я структури>]
{ <тип 1> ім'я поля 1;
<тип 2> ім'я поля 2 . . .; } p1, p2 . ..;
```

де **struct** — службове слово;

<ім'я структури> — ім'я типу структура (може бути відсутнім);

<тип 1>, <тип 2> — імена стандартних або визначених типів;

ім'я поля 1, ім'я поля 2, ... — імена полів структури;

p1, **p2** . . .; — імена змінних типу структура.

Наприклад, для опису даних про студента визначимо таку структуру:

```
struct stud
{ char name [25];          // ім'я
  char gender;             //стать
  int year;                // рік вступу
  double  rating;          // рейтинговий бал
} st1, st2;
```

Змінні **st1** і **st2** можна оголосити окремим оператором, наприклад:

```
stud st1, st2; .
```

Можна ініціювати поля структури або під час її описі, або в тілі програми. Під час опису структури ініціювання полів може виглядати так:

```
struct stud
{
   char name[25];
   char gender;
   int year;
   double rating;
} st1, st2;
st1 = {"Sidenko", 'm', 2022, 188.4};
st2 = {"Kononenko", 'w', 2024, 195.1};
```

Коли ми оголошуємо змінну структури **stud st1**, то **st1** посилається на всю структуру. Для того, щоб отримати доступ до окремих її членів, використовується оператор вибору члена (**.**):

```
        ім'я_структ_об'єкта. ім'я_поля;
```

Наприклад,

```
    st1.year = 2024;
    st2.rating = 197.4;
```

Розглянемо ілюстративну програму:

```
#include<iostream>
#include <windows.h> //для SetConsole()

using namespace std;
struct stud
{
        char name[25];
        char gender;
        int year;
        double rating;
};

int main()
{       SetConsoleOutputCP(1251);
        SetConsoleCP(1251);   //для українського шрифту
        stud a;          //оголошення змінної типу структура
        //введення значень в поля структури
        cout << "Введи прізвище: ";      cin.getline(a.name, 25);
        cout << "Введи стать 'ч/ж': ";   cin >> a.gender;
        cout << "Введи рік вступу: ";    cin >> a.year;
        cout << "Введи рейтинговий бал: "; cin >> a.rating;
        //обробка та виведення повідомлення
    if (a.gender == 'ж')
{cout << "\n Вона вступила в університет в " << a.year;
cout << " році з рейтинговим балом " << a.rating;}
     else
{cout << "\n Він вступив в університет в " << a.year;
cout << " році з рейтинговим балом " << a.rating;}
return 0;
}
```

Результат:

```
Введи прізвище: Кравченко
Введи стать 'ч/ж': ж
Введи рік вступу: 2025
Введи рейтинговий бал: 168.9

 Вона вступила в університет в 2025 році з рейтинговим балом 168.9
```

У наведеній програмі організується присвоювання всім полям структури *stud* відповідних значень. Далі виводиться повідомлення з використанням введеної інформації.

Якщо використовується тільки один структурний тип, то цей тип можна оголосити без імені. Тоді раніше розглянуту структуру можна оголосити таким чином:

```
struct
{
   char name[25];
   char gender;
   int year;
   double rating;
 } a;
```

У С ++ 11 додали можливість привласнювати членам структури значення за замовчуванням. Наприклад:

```
struct Rectangle
 {
     double length = 2.0;
     double width = 1.0;
 };

Rectangle z;  // довжина = 2.0, ширина = 1.0
z.length = 3.0;
// можна привласнювати членам структури і інші значення
```

Коли під час опису структур у деякій функції або в межах видимості змінних у різних функціях є багато (але не всі) однакових полів, то їх треба об'єднати в окрему структуру. Її можна застосовувати під час опису інших структур, тобто поля структури можуть самі бути типу *struct*. Це називається **вкладеністю структур** — її можна використати, наприклад, якщо треба обробляти списки студентів і викладачів університету. Студентські списки містять дані: прізвище та ініціали, дата (день, місяць, рік) народження, група та середній бал успішності, а в списках викладачів присутні такі дані: прізвище, ініціали, дата народження, кафедра, посада. У процесі обробки списку студентів і списку викладачів можна оголосити відповідно такі структури:

```
struct stud  {
  char name [25];
  int day, year;
  char month [10];
  char group[10];
  double rating; }
```

```
struct teacher {
  char name [25];
  int day, year;
  char month [10];
  char department[10];
  char job_title[15];}
```

В оголошених типах однакові поля можна об'єднати в окрему структуру та застосовувати її під час опису інших типів.

Поетапно це виглядає так:

– загальна структура:

```
struct general
{       char name[25];
        int day, year;
        char month[10];
}
```

– структура для опису інформації про студентів:

```
struct stud
{       general gn;
        char group;
        double rating
} st1, st2;
```

– структура для опису інформації про викладачів:

```
struct teacher
{       general gn;
        char department[10];
        char job_title[15];
} pr1, pr2;
```

У структурах ***stud*** і ***teacher*** для оголошення поля, що містить дані про прізвище та дату народження, використовується раніше описаний тип *general*. Тепер до поля *name, day, year, month* можна звернутися, використовуючи уточнюючий запис

ім'я_об'єкта. ім'я_об'єкта_вкладеної структури. ім'я_поля,

наприклад:

```
st1.gn.year = 2000;
pr2.gn.day = 15;
```

Після оголошення структурного типу змінних для роботи з їхніми полями можна застосовувати і покажчики, тоді опис структури матиме вигляд:

```
struct stud {
        char prizv[25];
        int mat, fiz, prg;
        double sb;
} st1, * pst;
```

Доступ до полів може здійснюватися двома способами:

– з використанням операції розіменування «*», тобто

```
cout<<(*pst).prizv;
(*pst).fiz = 65; /*розіменування покажчика має бути у круглих дужках, оскільки оператор вибору члена має вищий пріоритет, ніж оператор розіменування*/
```

– з використанням операції «**->**» для здійснення доступу до членів через покажчик, наприклад,

```
    cout << pst ->prizv;
    pst -> fiz = 65;
```

Крім того, до полів змінної *st1* можна звертатися, вказуючи поля через операцію «**.**», як це робилося раніше.

Дані типу структура можна об'єднати в масиви, наприклад, структуру можна записати так:

```
struct stud {
        char prizv[20];
        int mat, fiz, prg;
        float sb;
} spysok[15], * sp = &spysok[0];
```

У випадку, коли масив описується десь у тексті програми, тобто не саме після опису структури, його можна оголосити у вигляді:

stud spysok [15]; — масив типу структура з ім'ям *stud*, що містить відповідну інформацію про групу із 15 студентів.

Доступ до елементів масиву типу структура здійснюється із застосуванням індексу або через покажчик-константу, яким є ім'я масиву, тобто одним з таких способів:

```
    strcpy (spysok [1].prizv, " ");
    spysok [1].fiz = 65;
```

або

```
    strcpy ((sp + 1) ->prizv, " ");
    (sp + 1) -> fiz = 65;
```

або

```
    strcpy ((*(spysok + 1)).prizv, " ");
    (*(sp + 1)).fiz = 65;
```

У останньому виразі *(*(spysok + 1)).fiz = 65;* потрібна зовнішня пара дужок, тому що операція «**.**» («крапка») має пріоритет вище, ніж операція розіменування «***»**.

Поля структури можуть також бути масивами. Наприклад, у розглянутій структурі *stud* можна оцінки з різних предметів об'єднати в масив. Тоді структуру треба описати у вигляді:

```
struct stud1 {char prizv[25];
              int discipline[3];
              float sb;
} spysok[15], * ps = &spysok[0];
```

Звернення до полів здійснюватиметься одним із способів: *((*ps).prizv), (ps -> discipline[0])* , наприклад, *gets ((*ps).prizv);*
cin >> ps -> discipline [0] >> ps -> discipline [1] >> ps -> discipline [2]; або *cin >> ps ->*(discipline+0) >> ps ->*(discipline+1) >> ps ->*(discipline+2);*

Розглянемо використання структурного типу на прикладах.

Приклад 8.1.

Увести в комп'ютер відомість успішності групи студентів, які здали іспити з дисциплін «Математика», «Фізика» та «Програмування», та обчислити:

- середній бал кожного студента;
- середній бал за кожним предметом;
- вивести на екран прізвища відмінників з програмування.

Розглянемо перший варіант реалізації поставленої задачі, що може мати вигляд:

```
#include<iostream>
#include <windows.h> //для SetConsole()

using namespace std;
int main()
{
SetConsoleOutputCP(1251);
SetConsoleCP(1251);   //для українського шрифту

        const int n = 5;        // n – кількість студентів
        int   i;
        double sm, sf, sp;
//sm, sf, sp – сума оцінок групи відповідно з
//математики, фізики, програмування
struct stud
{
char prizv[25];
int mat, fiz, prg;
double sb;
} vid[n];        //  vid[n] – масив студентів (відомість)
        sm = sf = sp = 0;
//--введення та обробка інформації про студентів і їх успішність
        for (i = 0; i < n; i++)
        {
cout << "*****Введіть інформацію про " << (i + 1) << " студента\n";
        {
        cout << "Введіть прізвище ";
        cin >> vid[i].prizv;
        cout << "Оцінки з математики, фізики та програмування\n";
        cin >> vid[i].mat >> vid[i].fiz >> vid[i].prg;
```

```
//------- обчислення середнього балу студента за сесію
        vid[i].sb = (double(vid[i].mat + vid[i].fiz + vid[i].prg)) / 3;

//------ сума оцінок в групі по предметах
                        sm += vid[i].mat;
                        sf += vid[i].fiz;
                        sp += vid[i].prg;
                }
        }
// ---------------- виведення результатів обчислень

        cout << "\n******* Результати сесії\n";
        cout.precision(3);
        for (i = 0; i < n; i++)
{cout << i + 1 << "  " << vid[i].prizv;
cout << "\tматем. = " << vid[i].mat << "  фізика = " << vid[i].fiz;
cout << "  програм. = " << vid[i].prg;
cout << "  сер. бал = " << vid[i].sb << "\n";}
        cout << "\n\nСередній бал групи з математики = " << sm / n;
        cout << "\nСередній бал групи з фізики = " << sf / n;
        cout << "\nСередній бал групи з програмування = " << sp / n;
        cout << "\n\n***** Відмінники з програмування: \n";
        for (i = 0; i < n; i++)
                if (vid[i].prg >=90)  cout << vid[i].prizv << "\n";
return 0;
}
```

Результати виконання програми:

```
*****Введіть інформацію про 1 студента
Введіть прізвище Петренко
Оцінки з математики, фізики та програмування
69 78 68
*****Введіть інформацію про 2 студента
Введіть прізвище Шевченко
Оцінки з математики, фізики та програмування
88 95 99
*****Введіть інформацію про 3 студента
Введіть прізвище Костенко
Оцінки з математики, фізики та програмування
60 73 98
*****Введіть інформацію про 4 студента
Введіть прізвище Кошкіна
Оцінки з математики, фізики та програмування
80 96 95
*****Введіть інформацію про 5 студента
Введіть прізвище Манько
Оцінки з математики, фізики та програмування
92 90 90

******* Результати сесії
1  Петренко    матем. = 69  фізика = 78  програм. = 68  сер. бал = 71.7
```

```
2  Шевченко    матем. = 88  фізика = 95  програм. = 99  сер. бал = 94
3  Костенко    матем. = 60  фізика = 73  програм. = 98  сер. бал = 77
4  Кошкіна     матем. = 80  фізика = 96  програм. = 95  сер. бал = 90.3
5  Манько      матем. = 92  фізика = 90  програм. = 90  сер. бал = 90.7


Середній бал групи з математики = 77.8
Середній бал групи з фізики = 86.4
Середній бал групи з програмування = 90

***** Відмінники з програмування:
Шевченко
Костенко
Кошкіна
Манько
```

Наведемо другий варіант розв'язання поставленої задачі з використанням покажчиків та структуру, яка містить поле оцінок з предметів у вигляді масиву.

```
#include<iostream>
#include <windows.h> //для SetConsole()

using namespace std;
int main()
{
SetConsoleOutputCP(1251);
SetConsoleCP(1251);
const int n = 5, k = 3;
//n – кількість студентів, k – кількість оцінок
int   i, j;
double sm, sf, sp;
//sm, sf, sp – сума оцінок групи відповідно з математики, фізики,
//програмування

struct stud
{
        char prizv[25];
        int ocnk[3];                //   ocnk[3] – масив оцінок
        double sb;
} vid[n], * pst = &vid[0];     //   pst – покажчик на масив vid[n]
        sm = sf = sp = 0;
// введення та обробка інформації про студентів і їхня успішність
        for (i = 0; i < n; i++, pst++)
        {
cout << "Введіть інформацію про " << (i + 1) << " студента\n";
        { cout << "Введіть прізвище ";
        cin >> (*pst).prizv;
        cout << "Оцінки з математики, фізики та програмування\n";
// введення оцінок і розрахунок середнього бала студента
        (*pst).sb = 0;
```

```
        for (j = 0; j < 3; j++)
        {
                cin >> (*pst).ocnk[j];
                (*pst).sb += (*pst).ocnk[j];
        }
        (*pst).sb = (*pst).sb / 3; //середній бал студента за сесію

//------ сума оцінок в групі по предметах
                sm += (*pst).ocnk[0];
                sf += (*pst).ocnk[1];
                sp += (*pst).ocnk[2];
                }
        }
// ---------------------- виведення результатів обчислень
        pst -= n;   // pst =&vid[0]; – повернення покажчика початок масиву
        cout.precision(4);
        cout << "******* Результати сесії:\n";
        for (i = 0; i < n; i++, pst++)
{cout << i + 1 << "  " << (*pst).prizv;
cout <<"\tматем. = "<<(*pst).ocnk[0]<<" фізика = "<<(*pst).ocnk[1];
cout <<" програм. = "<<(*pst).ocnk[2]<<"  сер. бал = "<<(*pst).sb;
cout << "\n";}
        cout << "\n\nСередній бал групи з математики  = " << sm / n;
        cout << "\nСередній бал групи з фізики = " << sf / n;
        cout << "\nСередній бал групи з програмування = " << sp / n;
        cout << "\n\n*** Відмінники з програмування: \n";
        pst -= n;
for (i = 0; i < n; i++, pst++)
        if ((*pst).ocnk[2] >= 90)  cout << (*pst).prizv << "\n";
return 0;
}
```

Результат:

```
Введіть інформацію про 1 студента
Введіть прізвище Двінятін
Оцінки з математики, фізики та програмування
67 84 79
Введіть інформацію про 2 студента
Введіть прізвище Котенко
Оцінки з математики, фізики та програмування
89 68 92
Введіть інформацію про 3 студента
Введіть прізвище Мінько
Оцінки з математики, фізики та програмування
99 87 91
Введіть інформацію про 4 студента
Введіть прізвище Суліма
Оцінки з математики, фізики та програмування
89 92 82
Введіть інформацію про 5 студента
Введіть прізвище Кравченко
```

```
Оцінки з математики, фізики та програмування
80 90 93
******* Результати сесії:
1  Двінятін    матем. = 67  фізика = 84  програм. = 79  сер. бал = 76.7
2  Котенко     матем. = 89  фізика = 68  програм. = 92  сер. бал = 83
3  Мінько      матем. = 99  фізика = 87  програм. = 91  сер. бал = 92.3
4  Суліма      матем. = 89  фізика = 92  програм. = 82  сер. бал = 87.7
5  Кравченко   матем. = 80  фізика = 90  програм. = 93  сер. бал = 87.7


Середній бал групи з математики  = 84.8
Середній бал групи з фізики = 84.2
Середній бал групи з програмування = 87.4

*** Відмінники з програмування:
Котенко
Мінько
Кравченко
```

Треба нагадати, що запис виразу з покажчиком, наприклад, *(*pst).prizv*, рівнозначний запису *pst ->prizv*.

В останніх версіях C++ до складу структури можна включати не тільки поля (тобто дані), але і функції обробки полів (методи) (розділ 9). Це вдало ілюструє третій варіант розв'язання даного прикладу (середній бал групи з кожного предмета не обчислювався):

```
#include<iostream>
#include <windows.h> //для SetConsole()

using namespace std;
int main()
{
SetConsoleOutputCP(1251);
SetConsoleCP(1251);
const int n = 5;
int i;
//-------------------- опис структури
        struct Tstud
        {
                char prizv[15];
                int mat, fiz, prg;
                double sb()  // функція розрахунку середнього бала
        {
        double s = double(mat + fiz + prg) / 3;
        return s;
        }
                void input() // функція введення даних
        {
        cout << " Введіть прізвище\n";
        cin >> prizv;       //введення прізвищ без ініціалів
cout << "Введіть оцінки з математики, фізики, програмування\n";
```

```
        cin >> mat >> fiz >> prg;
        }
                void output()   // функція виведення даних
        {
cout << prizv << "\nматематика =" << mat << " фізика =" << fiz;
cout << " програмування =" << prg;
        }
                }vid[n]; //vid[n] – масив студентів (відомість)

// введення інформації
        for (i = 0; i < n; i++)
        {
        cout << " Введіть інформацію про " << (i + 1) << "  студента \n";
                vid[i].input();
        }
// виведення результатів
        cout << "******* Результати сесії \n";
        for (i = 0; i < n; i++)
        {
                vid[i].output();  cout.precision(3);
                cout << " середній бал =" << vid[i].sb() << endl;
        }
        cout << "\n*** Відмінники з програмування ***\n";
        int k = 0; //кількість відмінників з програмування
        for (i = 0; i < n; i++)
                if (vid[i].prg >= 90)
                     {
                      k++;
                      cout << vid[i].prizv << endl;
                      }
        if (k == 0) cout << " відсутні ";
return 0;
}
```

Результати:

```
Введіть інформацію про 1 студента
 Введіть прізвище
Рак
Введіть оцінки з математики, фізики, програмування
91 90 92
 Введіть інформацію про 2 студента
 Введіть прізвище
Машкова
Введіть оцінки з математики, фізики, програмування
76 82 82
 Введіть інформацію про 3 студента
 Введіть прізвище
Кремінь
Введіть оцінки з математики, фізики, програмування
79 92 95
 Введіть інформацію про 4 студента
```

```
 Введіть прізвище
Пустовіт
Введіть оцінки з математики, фізики, програмування
62 64 87
 Введіть інформацію про 5 студента
 Введіть прізвище
Зуєв
Введіть оцінки з математики, фізики, програмування
74 83 91
******* Результати сесії
Рак
математика =91 фізика =90 програмування =92 середній бал = 91
Машкова
математика =76 фізика =82 програмування =82 середній бал = 80
Кремінь
математика =79 фізика =92 програмування =95 середній бал = 88.7
Пустовіт
математика =62 фізика =64 програмування =87 середній бал = 71
Зуєв
математика =74 фізика =83 програмування =91 середній бал = 82.7

*** Відмінники з програмування ***
Рак
Кремінь
Зуєв
```

В такій реалізації вводиться додаткова змінна ***k*** для підрахунку кількості відмінників, якщо ***k = 0*** – виводитиметься повідомлення про їхню відсутність.

Наведемо приклад, в якому ініціювання масиву типу структура здійснюється під час опису, а в процесі роботи з масивом використовується той факт, що ім’я масиву є покажчиком на масив.

Приклад 8.2.

Увести з клавіатури таку інформацію про монітори: назву, термін гарантії, ціну. Відсортувати монітори за спаданням ціни та вивести відповідні повідомлення.

У програмі в рядках коментарів наведено інший можливий спосіб запису звернення до полів структури.

```
#include<iostream>
#include <windows.h> //для SetConsole()

using namespace std;
int main()
{
SetConsoleOutputCP(1251);
SetConsoleCP(1251);
```

```
const int n = 6;
struct Monitor
{
        char name[20];  // марка монітора
        int garant;     // термін гарантії
        int cina;       // ціна
} mon[n] = {" Asus PA279CV ", 36, 17802,
            " Dell U2421E ", 36, 12539,
            " Philips 27M1F58 ", 24, 27899,
            " Asus PG48UQ ", 12, 67999,
            " Samsung G95NC ", 36, 84999,
            " Eizo EV3240X-BK ", 36, 83659 };
        //  ініціалізація масиву mon[n] при описі
                int i, k;
cout << "\t ***Список моніторів (модель, гарантія, ціна)***\n";

        //  виведення інформації про монітори
                for (i = 0; i < n; i++)
                {
cout << (*(mon + i)).name<< " ";    // cout << (mon + i) -> name;
cout << (*(mon + i)).garant<< " "; // cout <<  (mon + i) -> garant;
cout << (*(mon + i)).cina << "\n"; // cout <<(mon + i)->cina<<"\n";
                }
// сортування за зменшенням ціни
        Monitor temp; //temp – для перестановки елементів масиву

        for (k = 1; k < n; k++)
                for (i = 0; i < n - k; i++)
                if ((*(mon + i)).cina < (*(mon + i + 1)).cina)
                        //  if ((mon + i) -> cina < (mon + i+1) -> cina)
                                {
                                temp = *(mon + i);
                                *(mon + i) = *(mon + i + 1);
                                *(mon + i + 1) = temp;
                                }

        cout << "\n\n\t *******  Сортування за ціною  *******\n";

        /*-------- виведення інформації за зменшенням ціни   */
                for (i = 0; i < n; i++)
                {
        cout << "ціна (грн): ";
        cout << (*(mon + i)).cina;       //  cout << (mon + i)-> cina;
        cout << " модель ";
        cout << (*(mon + i)).name<<"     "; // cout<< (mon + i)-> name;
        cout << "\tгарантія (місяців): ";
        cout << (*(mon + i)).garant <<"\n";//cout<<(mon+i)->garant;
                }
return 0;
}
```

Результати:

```
        ***Список моніторів (модель, гарантія, ціна)***
Asus PA279CV 36 17802
Dell U2421E 36 12539
Philips 27M1F58 24 27899
Asus PG48UQ 12 67999
Samsung G95NC 36 84999
Eizo EV3240X-BK 36 83659


        *******  Сортування за ціною  *******
ціна (грн): 84999 модель  Samsung G95NC        гарантія (місяців): 36
ціна (грн): 83659 модель  Eizo EV3240X-BK      гарантія (місяців): 36
ціна (грн): 67999 модель  Asus PG48UQ          гарантія (місяців): 12
ціна (грн): 27899 модель  Philips 27M1F58      гарантія (місяців): 24
ціна (грн): 17802 модель  Asus PA279CV         гарантія (місяців): 36
ціна (грн): 12539 модель  Dell U2421E          гарантія (місяців): 36
```

Приклад 8.3.

Обробити інформацію про групу студентів, що містить прізвища, атрибут «бюджет/контракт» і середній бал успішності. Вивести повідомлення, чий підсумковий середній бал вищий — бюджетників чи контрактників.

Перший варіант програми розв'язання прикладу передбачає, що признак «бюджет/контракт» студента при введенні інформації позначається літерами «*k*» та «*b*».

```
#include<iostream>
using namespace std;
int main()
{
        const int n = 6;
        struct {
                char name[15];    // ім'я
                char attribute;   // признак «бюджет/контракт»
                float avr;        // середній бал
        }  stud[n];               // масив студентів

        float  avr_k = 0, avr_b = 0;
// avr_k, avr_b – середній бал відповідно контрактників і бюджетників
        int  i, k = 0, b = 0;
// k, b – лічильники кількості контрактників і бюджетників
        // введення початкової інформації
        for (i = 0; i < n; i++)
        {
         cout << "Input name" << endl;  //   введення імені
         cin >> stud[i].name;
        // cin >> (*(stud+i)).name;  cin >> (stud+i)->name;
         cout << "Input attribute (k/b)" << endl;
```

```
         cin >> stud[i].attribute;
        //cin >> (*(stud+i)).attribute;  cin >> (stud+i)->attribute;
         cout << "Input average" << endl;
         cin >> stud[i].avr;
        //cin  >> (*(stud+i)).avr;    cin >> (stud+i)->avr;
        }

        for (i = 0; i < n; i++)
                if (stud[i].attribute == 'k')
//if((*(stud+i)).attribute=='k');
//if((stud+i)-> attribute=='k');

/* визначення кількості контрактників в групі та їх сумарного
середнього бала*/
        {
                k++;
                avr_k += stud[i].avr;
        //avr_k+=(*(stud+i)).avr;  avr_k+=(stud+ i)->avr;
        }
                else
/* визначення кількості бюджетників у групі та їхнього сумарного
середнього бала */
if (stud[i].attribute == 'b')  // if((*(stud +i)). attribute=='b');
                                //if((stud+ i)->attribute=='b');
        {
                        b++;
                avr_b += stud[i].avr;
                //avr_b+= (*(stud+i)).avr; avr_b+=(stud+ i)->avr;
        }
        cout.precision(4);
        cout << "budget students average score = " << avr_b / b;
        cout << "\ncontract students average score = " << avr_k / k;
if (avr_k / k > avr_b / b)
cout << "\nAmong contract students, the average score is higher";
        else
cout << "\nAmong budget students, the average score is higher";
return 0;
}
```

Результат:

```
Input name
Ken
Input attribute (k/b)
k
Input average
77.5
Input name
Nana
Input attribute (k/b)
b
```

```
Input average
81.4
Input name
Ted
Input attribute (k/b)
b
Input average
62.7
Input name
Pol
Input attribute (k/b)
k
Input average
82.2
Input name
Sati
Input attribute (k/b)
b
Input average
72.8
Input name
Lila
Input attribute (k/b)
k
Input average
84.6
budget students average score = 72.3
contract students average score = 81.43
Among contract students, the average score is higher
```

У рядках коментарів програми наведено інший спосіб запису звернення до полів структури для випадку, коли ім'я масиву використане як покажчик на масив.

Другий варіант розв'язання прикладу містить змінну-покажчик та функцію порівняння рядків. При введенні інформації признак форми фінансування студента позначається рядками символів відповідно «budget» та «contract».

```
#include<iostream>
#include<string.h>
using namespace std;
const int n = 6;

int main()
{
```

```
struct stud
{
                char name[20];
                char attribute[10];
                float avr;
};
        stud a[n], * p(&a[0]);
        float avr_k, avr_b;
        int k, b, i;
        //введення інформації
        for (i = 0; i < n; i++, p++)
        {
                cout << "Input name" << endl;
                cin >> (*p).name;       // cin >> p->name;
                cout << "Input attribute (budget/contract)" << endl;
                cin >> (*p).attribute;     // cin >> p->attribute;
                cout << "Input average" << endl;
                cin >> (*p).avr;        // cin >> p->avr;
        }

        p -= n; //покажчик повертається на початок масиву
        k = b = avr_k = avr_b = 0;
        for (i = 0; i < n; i++, p++)
                //  функція strcmp() порівнює 2 рядки символів
                if (strcmp((*p).attribute, "contract") == 0)
                        //if(strcmp(p->attribute,"contract")==0)
                {
                        k++;
                        avr_k += (*p).avr; // avr_k+= p->avr;
                }
                else
                        if (strcmp((*p).attribute, "budget") == 0)
                        //if(strcmp(p->attribute,"budget")==0)
                        {
                                b++;
                                avr_b += (*p).avr;
                        }
        cout.precision(4);
        cout << "budget students average score = " << avr_b / b;
        cout << "\ncontract students average score = " << avr_k / k;
        if (avr_k / k > avr_b / b)
cout << "\nAmong contract students, the average score is higher";
        else
cout << "\nAmong budget students, the average score is higher";
        return 0;
}
```

Результати роботи програми такі ж, як і попередньої. У програмі для реалізації функції порівняння рядків *strcmp()* використана бібліотека *string.h.*

8.2 Запитання та завдання

1. Як описуються дані типу структура? Наведіть приклади використання структур.
2. Які типи полів може містити структура?
3. Поясніть призначення функцій обробки полів.
4. Які існують способи ініціювання полів структури?
5. Що таке вкладеність структур?
6. Як здійснюється робота з типом *struct* з використанням покажчиків?
7. Як реалізується доступ до елементів масиву типу структура?

Завдання.

Набуття практичних навичок програмної реалізації задач з даними типу структура під час розв'язання завдань розділу вправ «Структури».

9 ФУНКЦІЇ

9.1 Поняття функції

Функція — це іменована логічно завершена сукупність оголошень і операторів, призначених для виконання певної задачі [1, 4 — 8, 12 — 14, 18, 20].

Програма мовою С++ містить одну або декілька функцій, кожна з яких має бути оголошена та визначена до її першого використання. **Оголошення функції** (**прототип**, **заголовок**) задає ім'я функції, тип значення, що повертає функція (якщо воно є), а також імена та типи аргументів, які можуть передаватися як у функцію, так і з неї. **Визначення функції** — це задання способу виконання операцій.

Треба нагадати, що серед функцій програми має бути одна з ім'ям **main** (**головна функція**), яка може знаходитися в будь-якому місці програми. Ця функція виконується завжди першою та закінчується останньою. Усі функції мають однакову структуру визначення у вигляді:

```
[тип результату] ім'я функції ([список формальних аргументів])
{// тіло функції
 опис даних;   оператори;
[return] [вираз] ;},
```

де **тип результату** — будь-який базовий або раніше описаний тип значення (за винятком масиву та функції), що повертається функцією (необов'язковий параметр). За відсутності цього параметра тип результату за замовчуванням буде цілий (**int**). Він також може бути описаний ключовим словом (**void**), тоді функція не повертає ніякого значення. Якщо результат повертається функцією, то в тілі функції є необхідним оператор

```
return вираз;
```

де **вираз** формує значення, що співпадає з типом результату; **ім'я функції** — ідентифікатор функції, за яким завжди знаходиться пара круглих дужок «**( )**», де записуються **формальні аргументи.** Фактично **ім'я функції** — це особливий вид покажчика на функцію, його значенням є адреса початку входу у функцію;

список формальних аргументів — визначає кількість, тип і порядок проходження переданих у функцію вхідних аргументів, які розділяються комою («**,**»). У випадку, коли параметри відсутні, дужки залишаються порожніми або містять ключове слово (**void**). Формальні параметри функції локалізовані в ній і недоступні для будь-яких інших функцій.

Список формальних аргументів має такий вигляд:

```
([const] тип1 [параметр1], [const] тип2 [параметр2],...)
```

У списку формальних аргументів для кожного параметра треба вказати його тип (не можна групувати параметри одного типу, вказавши їхній тип один раз).

Тіло функції може складатися з описів змінних і операторів. Змінні, що використовуються під час виконання функції, можуть бути глобальні та локальні. Змінні, що описані (визначені) за межами функції, називають ***глобальними***. За допомогою глобальних параметрів можна передавати дані у функцію, не включаючи ці змінні до складу формальних параметрів. У тілі функції їх можна змінювати та потім отримані значення передавати в інші функції.

Змінні, що описані у тілі функції, називаються ***локальними*** або автоматичними. Вони існують тільки під час роботи функції, а після реалізації функції система видаляє локальні змінні та звільняє пам'ять. Тобто між викликами функції вміст локальних змінних знищується, тому ініціювання локальних змінних треба робити щоразу під час виклику функції. За необхідності збереження цих значень, їх треба описати як статичні за допомогою службового слова **static**, наприклад:

```
static int x, y;
```

або

```
static float p = 3.25; .
```

Статична змінна схожа на глобальну, але діє тільки у тій функції, в якій вона оголошена.

На початку програми можна не описувати всю функцію, а записати тільки **прототип**. Запис прототипу може містити тільки перелік типів формальних параметрів без імен, а наприкінці прототипу завжди ставиться символ «**;**», тоді як у описі (визначенні) функції цей символ після заголовка не присутній.

Механізм передачі параметрів є основним засобом обміну інформацією між функцією, що викликається, та функцією, яка викликає. Параметри, котрі зазначаються у заголовку опису функції, як відомо, називаються **формальними**, а параметри, які записані у операторах виклику функції — **фактичними**. Наведемо приклад фрагмента програми з використанням функцій:

```
#include<iostream>
using namespace std;

double qwadro(double);  // прототип функції qwadro()
int main()              // головна функція
```

```
{
int n;
cout << "Input n = ";
cin >> n;       //введення фактичного значення
cout << "n x n = " << qwadro(n); //виклик функції qwadro()
return 0;
}
double qwadro(double p) // функція qwadro()
{
return p * p;           // повернення значення
}
```

У результаті виконання програми буде виведено:

```
Input n = 233
n x n = 54289
```

Функція завжди має бути визначена або оголошена до її виклику. ***При оголошенні, визначенні та виклику тієї самої функції типи та послідовність параметрів повинні співпадати***. На імена параметрів обмежень на відповідність не існує, оскільки функцію можна викликати з різними аргументами, а в прототипах імена ігноруються компілятором (вони необхідні тільки для покращення читання програми). Тип значення, що повертає функція, та типи параметрів спільно визначають тип функції.

У найпростішому випадку при виклику функції необхідно вказати її ім'я, за яким у круглих дужках через кому перелічити імена аргументів, що передаються. Виклик функції може здійснюватися у будь-якому місці програми, де за синтаксисом дозволяється вираз того типу, що формує функція. Якщо тип значення, що повертає функція не **void**, вона може входити до складу виразів або, у поодинокому випадку, розташовуватися у правій частині оператора присвоювання.

Виконання виклику функції виконується таким чином:

1. Обчислюються вирази в списку фактичних параметрів і піддаються звичайним арифметичним перетворенням. Потім, якщо відомий прототип функції, тип отриманого фактичного аргументу порівнюється з типом відповідного формального параметра. Якщо вони не збігаються, то або виробляється перетворення типів, або формується повідомлення про помилку. Число виразів у списку виразів має збігатися з числом формальних параметрів, якщо тільки функція не має змінного числа параметрів. В останньому випадку перевірці підлягають тільки обов'язкові параметри. Якщо в прототипі функції зазначено, що їй не потрібні параметри, а при виклику вони вказані, формується повідомлення про помилку.

2. Відбувається привласнення значень фактичних параметрів відповідним формальним параметрам.

3. Управління передається на перший оператор функції.

4. Виконання оператора **return** в тілі функції повертає управління і можливо, значення в викликаючу функцію. За відсутності оператора **return** управління повертається після виконання останнього оператора тіла функції, а значення, що повертається, не визначене.

У мові С++ визначено декілька способів передачі параметрів та повернення результату обчислень функцій, серед них найбільш широке використання набули:

– виклик функції з передачею параметрів за допомогою формальних аргументів значень;

– виклик функції з передачею адрес за допомогою параметрів-покажчиків;

– виклик функцій з використанням посилань, коли доступ до переданих параметрів забезпечується за допомогою альтернативного імені (синоніма);

– виклик функцій з передачею даних за допомогою глобальних змінних;

– виклик функцій із застосуванням параметрів, що задані за замовчуванням, при цьому можна використовувати або всі аргументи, або їхню частину.

Виклик функції з передачею значень полягає у тому, що у функцію передаються не самі аргументи, а їхні копії. Ці копії можна змінювати всередині функції, і це ніяк не позначиться на значеннях аргументів, що за межами функції залишаться без зміни, наприклад:

```
#include<iostream>
using namespace std;
void fun(int p)          // визначення функції fun()
{       ++p;             //збільшуємо значення параметру на 1
        cout<<"p = " << p << endl;
}
int main()      // головна функція
{
        int x = 10;
        fun(x); // виклик функції
        cout << "x = " << x;
return 0;
}
```

Результат:

```
p = 11
x = 10
```

Це пояснюється наступним чином. При виклику функції *fun(x)* до неї передається копія значення, що дорівнює 10. Всередині функції значення копії змінної збільшується на 1, тобто *(++p)*, і тому

виводиться *p = 11*, але за межами функції параметр *p* не змінюється. У цьому випадку функція не повертає ніякого значення.

При цьому способі для звернення до функції достатньо написати її ім'я, а в дужках — значення або перелік фактичних аргументів. Фактичні аргументи повинні бути записані в тій же послідовності, що і формальні, та мати відповідний тип (крім аргументів за замовчуванням і перевантажених функцій).

Якщо формальними аргументами функції є параметри значення та в ній не використовуються глобальні змінні, функція може передати у викликаючу її функцію лише одне значення, що записується в операторі **return**. Це значення передається в місце виклику функції. Достроковий вихід з функції можна також організувати з використанням оператора **return**.

За необхідності зміни параметра за межами тіла функції можна передати значення цього параметра за допомогою **параметра-покажчика** або **за посиланням**.

Виклик функції з використанням покажчиків забезпечує передачу до функції не значень параметрів, а їх адреси, що дозволяє міняти значення цих змінних усередині функції та передавати за її межі (в інші функції).

У цьому випадку для виклику функції у списку формальних параметрів необхідно записати адресу того параметра, який треба змінити, тоді відповідний формальний параметр матиме тип покажчика на цей параметр. Усередині функції здійснюється розіменування параметра покажчика та виконання необхідних дій. Програма з використанням виклику функції з передачею адрес за допомогою параметрів – покажчиків може мати вигляд:

```
// використання параметра покажчика
#include<iostream>
using namespace std;
void fun(int*p)          // визначення функції fun()
{
        ++*p;            //збільшуємо значення параметру на 1
        cout<<"*p = " << *p<<endl;
}
int main()      // головна функція
{
        int x = 10;
        fun(&x);// виклик функції
        cout << "x = " << x ;
return 0;
}
```

У результаті буде виведена інформація:

```
*p = 11
x = 11
```

Виклик функції з використанням параметра-посилання здійснює передачу до функції не самої змінної, а тільки посилання на неї.

У загальному випадку можна знайти приблизно таке визначення: "Посилання – це альтернативне ім'я, це псевдонім об'єкта" та інші визначення, в яких йдеться про те, що посилання – це інше ім'я об'єкта.

Для створення посилання використовують знак «**&**», який застосовується до типу:

```
тип & ім’я_посилання = ім’я_змінної;
```

Створення змінної посилального типу – це не створення змінної, а створення нового ідентифікатора під якийсь об'єкт, найчастіше вже іменований.

На відміну від звичайних змінних посилання вимагають **негайної ініціалізації себе** об'єктом.

```
// Повноцінне створення посилання
int value;  // об'єкт, існуючий до створення посилання
int& ref = value;
// створення посилання ref і ініціалізація його об'єктом value
```

Розглянемо приклад:

```
#include<iostream>
using namespace std;
int main()
{       int x = 10;
        int& ref = x;           //створення посилання
        cout << ++x<<endl;      //11
        ref++;
        cout << x << endl;      //12
        cout << ref << endl;    //12
        cout << &x << endl;     //адреса x BEFD4FF6B4
        cout << &ref;           //та ж сама адреса BEFD4FF6B4
return 0;
}
```

Будь-яка зміна шляхом використання посилання впливає не на посилання, а на об'єкт, до якого одного разу було прив'язане посилання.

Тип посилання має збігатися з типом об'єкта, до якого посилання прив'язується. Є два винятки (при поліморфізмі, при неявних приведеннях тип не повинен збігатися точно).

Зміна об'єкта нібито впливає на посилання, це ілюзія, посилання — це ідентифікатор цього самого об'єкта.

При передачі параметрів за посиланням передається посилання на об'єкт, через яке можна маніпулювати самим об'єктом, а не просто його значенням. Тоді всі дії, що відбуваються над посиланням, є діями над самою змінною. Такий спосіб передачі параметрів і повернення

результату передбачає запис у списку фактичних параметрів імені змінної, а у списку формальних — параметрів посилань.

Наприклад:

```
#include<iostream>
using namespace std;

// використання параметра посилання
void fun(int& p)// функція fun()
{
        ++p;
        cout << "p = " << p << endl;
}
int main()
{
        int x = 10;
        fun(x); // виклик функції fun() з параметром-посиланням
        cout << "x = "  << x;
return 0;
}
```

Результат:

```
p = 11
x = 11
```

Передача параметра за посиланням означає, що копіюється не саме значення, а адреса вихідної змінної (як у випадку передачі параметра за адресою), проте синтаксис використовується такий, щоб програмісту не доводилося використовувати операцію розіменування і він міг мати справу безпосередньо зі значенням, що зберігається за цією адресою (як у випадку передачі параметра за значенням). Коли необхідно, щоб деякі параметри не змінювали свої значення всередині функції, їх треба оголосити як параметри константи, використовуючи модифікатор const.

Передача за посиланням дозволяє повернути з функції відразу кілька значень через параметри виведення. Наприклад, наступна програма демонструє використання функції, яка як параметри має 2 цілих числа, 2 посилання на ціле (в них розраховуватимуться сума та добуток першого та другого параметра) та посилання на дійсний тип *double* для збереження результату ділення першого параметра на другий (перевірка ділення на нуль в програмі відсутня):

```
#include <iostream>
using namespace std;

void f (int a, int b, int& sum, int &mult, double& div) {
    // sum - посилання для збереження суми
    // mult - посилання для збереження добутку
```

```
    // div посилання для збереження результату ділення
    sum = a + b;
    div= (double)a/b;
    mult = a * b;
}

int main() {
    int S = 1, M = 0;
    double D = 0;   //оголошення змінних для передачі їх у функцію
//ініціалізація не обов'язкова
    int x, y;
    cout << "Input value x = "; cin >> x;
    cout << "Input value y = "; cin >> y;
    f(x, y, S, M, D); //виклик функції
    cout << "sum x + y = " << S; //значення змінних змінилися
    cout << "\nmult x * y = " << M;
    cout << "\ndiv x / y = " << D;
return 0;
}
```

Результат роботи може бути таким:

```
Input value x = 11
Input value y = 45
sum x + y = 56
mult x * y = 495
div x / y = 0.244444
```

Використовувати глобальні змінні для передачі даних між функціями дуже легко, оскільки вони видимі в усіх функціях, де описані локальні змінні з тими ж іменами. Але такий спосіб не є поширеним, тому що може бути небезпечним, оскільки будь-яка функція може змінити значення глобальних змінних, це ускладнює налагодження програми та перешкоджає розташуванню функції у бібліотеці загального використання. Необхідно прагнути, щоб функції були максимально незалежними, а їхній інтерфейс повністю визначався прототипом функції. Наведемо приклад використання глобальних змінних:

```
#include <iostream>
using namespace std;
int a, b, c;    // глобальні параметри
void sum();     // прототип функції
int main()      // головна функція
{
        cin >> a >> b;
        sum();  // виклик sum()
        cout << c <<endl;
return 0;
}
void sum()       // функція sum()
```

```
{
        c = a + b;
}
```

В останніх версіях мови С++ з'явилася можливість **передавати дані за замовчуванням.** У цьому випадку під час написання функції всім аргументам або декільком з них присвоюються початкові значення та задовольняються такі вимоги: коли якому-небудь аргументу присвоєно значення за замовчуванням, то всі аргументи, що розташовані за ним (тобто записані праворуч), повинні мати значення за замовчуванням. Таким чином, список параметрів поділяється на дві частини: параметри, що не мають значення за замовчуванням, і параметри, що мають такі значення.

У випадку виклику функції для параметрів, що не мають значень за замовчуванням, обов'язково повинен бути фактичний аргумент, а для параметрів, що мають значення за замовчуванням, фактичні аргументи можна опускати, коли ці значення не треба змінювати.

Якщо деякий параметр має значення за замовчуванням та для нього відсутній фактичний аргумент, то і для всіх наступних (тобто записаних пізніше) параметрів фактичні аргументи повинні бути відсутні, тобто їхні значення передаються до функції за замовчуванням, наприклад:

```
#include <iostream>
using namespace std;

void fun(float x, int y = 25, int z = 100)
{
        cout << "x = " << x << " y = " << y << " z = " << z << endl;
}

int main()
{
fun(0.44, 3, 18); //за замовчуванням нічого не передається
fun(5.1, 10); // за замовчуванням передається один аргумент – z
fun(10.2);   // за замовчуванням передаються два аргумента – y, z
return 0;
}
```

На екрані буде виведено:

```
x = 0.44 y = 3 z = 18
x = 5.1 y = 10 z = 100
x = 10.2 y = 25 z = 100
```

З цієї ілюстративної програми видно, що аргумент за замовчуванням — це той аргумент, значення якого задане в ході опису заголовка функції, а при її виклику його можна не вказувати.

Якщо замість параметра, заданого за замовчуванням при звертанні до функції, записується інше значення фактичного параметра, то значення за замовчуванням перекривається заданим фактичним значенням. Так, наприклад, в останньому програмному фрагменті при виклику функції *fun (13.5, 75);* на екрані буде виведено: *x = 13.5 y = 75 z = 100,* тобто значення *z* — прийнято за замовчуванням.

Приклад 9.1.

Обчислити квадратну функцію вигляду $y = ax^2 + bx + c$ для заданого значення аргументу ***x*** з використанням функції, в яку коефіцієнти ***a***, ***b*** і ***c*** можуть бути введені за замовчуванням.

```
#include<iostream>
using namespace std;
//прототип функції з параметрами за замовчуванням
double equation(double x, double a = 0., double b = 0., double c = 0.);

int main()
{
        double a = 1., b = 2., c = 3., x = 0.5, y;

        cout << "\n***** All arguments entered " << endl;
        y = equation(x, a, b, c);    //----------виклик функції
        cout << " y = " << y << endl;

        cout << "\n***** Arguments x, a, b entered  " << endl;
        y = equation(x, a, b);
        cout << " y = " << y << endl;

        cout << "\n***** Argument x entered  " << endl;
        y = equation(x);
        cout << " y = " << y << endl;

return 0;
}

// функція обчислення квадратної залежності
double equation(double x, double a, double b, double c)
{
        return a* x* x + b * x + c;
}
```

Результати виконання програми:

```
***** All arguments entered
 y = 4.25

***** Arguments x, a, b entered
 y = 1.25
```

```
***** Argument x entered
 y = 0
```

Висновки відносно використання функцій:

– параметри функції, що передаються до неї за значенням, за межами функції не змінюються, тобто їх не можна використовувати для передачі результату роботи функції;

– передача результату функції за її межі здійснюється або за посиланням «**&**», або з використанням покажчика «*****», або за допомогою глобальних параметрів чи оператора **return**;

– локальні змінні використовуються в тілі функції, існують тільки під час роботи функції, а при виході з неї знищуються, тому такі змінні називаються автоматичними змінними та їх можна використовувати тільки для перетворень всередині функції;

– якщо виникає необхідність збереження значень локальних змінних між викликами функції, то вони повинні бути оголошені як статичні, тобто з описом **static**, наприклад:

```
static char st[] = "Тесленко А. М."; .
```

9.2 Масиви як параметри функцій

Аргументами (параметрами) функцій можуть бути не тільки змінні, але й масиви. У цьому випадку можна використовувати як масиви фіксованого розміру, так і невизначеного (масиви змінної довжини). В процесі застосування масивів фіксованої довжини в заголовку функції в списку формальних аргументів указується тип масиву та його розмір, наприклад:

```
    void sort(int mas[30]);.
```

Якщо описується функція з масивом змінної довжини, то в заголовку вказується тип масиву невизначеного розміру та обов'язково ще один параметр, за допомогою якого задається розмірність масиву, наприклад:

```
    void sort(int mas[], int n); .
```

Всі масиви у функції передаються за адресою (як покажчики), тому у випадку зміни масивів у функції ці зміни зберігаються при поверненні у функцію.

Наведемо приклад **використання глобальних змінних** для передачі даних у процесі роботи функції.

Приклад 9.2.

У масивах ***m1(10), m2(15), m3(12)*** визначити мінімальний елемент та його індекс.

```
#include<iostream>

using namespace std;


int ind = 0;   // глобальна змінна
//------функція введення елементів масиву
void input(double ar[], int n)
{
        for (int i = 0; i < n; i++)
        {
                cout << "Input " << i << " element ";
                cin >> ar[i];
        }
        cout << endl;
}
//-----функція визначення мінімального елемента масиву
double fmin(double ar[], int n)
{
        ind = 0;
        float min = ar[0];
        for (int i = 1; i < n; i++)
                if (ar[i] < min)
                {
                        min = ar[i];
                        ind = i;
                }
        return min;
}
//---------головна функція
int main()
{
        double m1[5], m2[6], m3[7];

        cout << "***** Input array m1 \n";
        input(m1, 5);   // виклик функції введення масиву m1[]
        //  виклик функції fmin()
        cout << "min1 = " << fmin(m1, 5);

        cout << " index = " << ind << endl;

        cout << "***** Input array m2\n";
        input(m2, 6);   // введення масиву m2[]
        cout << "min2 = " << fmin(m2, 6);
        cout << " index = " << ind << endl;
        cout << "***** Input array m3\n";
        input(m3, 7);   // введення масиву m3[]
        cout << " min3 = " << fmin(m3, 7);
        cout << " index = " << ind << endl;
return 0;
}
```

Результати обчислень:

```
***** Input array m1
Input 0 element 4.3
Input 1 element 2.6
Input 2 element 1.9
Input 3 element 5.8
Input 4 element -2.3

min1 = -2.3   index = 4
***** Input array m2
Input 0 element 2.2
Input 1 element -6.4
Input 2 element 2.9
Input 3 element -3.8
Input 4 element -9.4
Input 5 element 2.1

min2 = -9.4   index = 4
***** Input array m3
Input 0 element 2.6
Input 1 element 1.1
Input 2 element 0.5
Input 3 element 2.8
Input 4 element 8.4
Input 5 element 3.6
Input 6 element 7.2

 min3 = 0.5   index = 2
```

Програма, крім головної функції ***main()***, має також функцію ***input()*** введення елементів деякого формального масиву та функцію ***fmin()*** визначення мінімального елемента цього масиву. У головній функції здійснюється виклик функцій для розв'язання необхідних обчислень кожного конкретного масиву, для передачі параметрів використовується глобальна змінна ***ind***.

Під час обробки символьних масивів функцією кількість елементів можна не вказувати окремим параметром за рахунок того, що ознакою кінця рядка може виступати завершальний символ '\0'.

Приклад 9.3.

Навести приклад програмної реалізації, в якій відбувається передача символьного масиву у функцію. В цій функції розраховується довжина рядка, як параметр – масив символів, кількість елементів якого можна розрахувати за умови досягнення завершального символу '\0'.

```
#include<iostream>

using namespace std;

int len(char st[]);
int main()
{
        char p[] = "This is a string";
        int k = len(p);
        cout << "string length = " << k;
return 0;
}

int len(char st[])
{
        int n = 0;
        while (st[n]!= '\0') n++;
return n;
}
```

Результат виконання програми:

```
string length = 16
```

Оскільки ім'я масиву – покажчик, пов’язаний з початком масиву, то будь-який масив, що передається як параметр, може бути змінений за рахунок виконання тіла функції.

Приклад 9.4.

Реалізувати функції введення, виведення масиву та функцію, яка міняє місцями мінімальний та максимальний елементи масиву. Викликати ці функції для масивів різної розмірності.

Алгоритм знаходження максимального та мінімального елементів масиву розглянуто у прикладі 1.3.

```
#include<iostream>
using namespace std;

void input(int*, int);  //прототип функції введення масиву
void output(int*, int); //прототип функції виведення масиву
void change(int*, int); //прототип функції зміни масиву - мінімальний
                        //та максимальний елементи помінялися місцями
// головна функція
int main()
{
        int a[10], b[15];
        input(a, 10);
        cout << "Array a[10] before rearrangement:" << endl;
        output(a, 10);
        change(a, 10);
```

```
        cout << "Array a[10] after rearrangement:" << endl;
        output(a, 10);
        input(b, 15);
        cout << "Array b[15] before rearrangement:" << endl;
        output(b, 15);
        change(b, 15);
        cout << "Array b[15] after rearrangement:" << endl;
        output(b, 15);
return 0;
}

//визначення функцій

void change(int* x, int k)
{
        int min = x[0], max = x[0];
        int i_min = 0, i_max = 0;
        for (int i = 1; i < k; i++)
                if (x[i] < min) {
                        min = x[i]; i_min = i;
                }
                else
                        if (x[i] > max) {
                                max = x[i]; i_max = i;
                        }
        x[i_min] = max; x[i_max] = min;
}
void input(int* x, int k)
{
        for (int i = 0; i < k; i++)
                x[i] = rand() % 100 - 50;
}
void output(int* x, int k)
{
        for (int i = 0; i < k; i++)
                cout << x[i] << " "; cout << endl;
      }
```

Результати обчислень:

```
Array a[10] before rearrangement:
-9 17 -16 -50 19 -26 28 8 12 14
Array a[10] after rearrangement:
-9 17 -16 28 19 -26 -50 8 12 14
Array b[15] before rearrangement:
-45 -5 31 -23 11 41 45 -8 -23 -14 41 -46 -48 3 42
Array b[15] after rearrangement:
-45 -5 31 -23 11 41 -48 -8 -23 -14 41 -46 45 3 42
```

Як видно, в результаті роботи функції масив змінився (синім кольором виділені елементи, які помінялися місцями).

Параметрами функцій можуть бути не тільки одновимірні, але й багатовимірні масиви. У цьому випадку використовуються масиви як фіксованої розмірності, так і невизначеної довжини.

У заголовку функції під час роботи з багатовимірним масивом фіксованого розміру, наприклад матриці *mat[7][10]*, вказуються обидві розмірності масиву, тобто:

```
void fun (int mat [7][10]); .
```

Якщо застосовується багатовимірний масив невизначеної довжини, то невизначеним може бути тільки один вимір розмірності (завжди перший!), наприклад:

```
void fun2 (int mat [ ][10], int rows, int cols); .
```

Приклад 9.5.

Для заданої матриці зробити обчислення середнього значення кожного її стовпця з використанням функції введення розмірності матриці, функції введення матриці та функції одержання середнього значення її стовпців.

```
#include<iostream>
using namespace std;

const int mincol = 2;   //мінімальна кількість стовпців
const int maxcol = 20;  //максимальна кількість стовпців
const int minrow = 2;   //мінімальна кількість рядків
const int maxrow = 30;  //максимальна кількість рядків

//функція getnum ( ) для введення кількості рядків і стовпців
int getnum(const char* elemtype, int low, int high)
{
        int  n;
        do
        {
        cout << "Enter the number of " << elemtype
                << " n from [" << low << "] to [" << high << "] : ";
                cin >> n;
        } while (n<low || n>high);
        return n;
}

//функція  inmatr( ) введення елементів матриці
void inmatr(double matr[][maxcol], int rows, int cols)
{
        for (int i = 0; i < rows; i++)
        {
```

```
                cout << "Enter " << i << " matrix row " << endl;
                for (int j = 0; j < cols; j++)  cin >> matr[i][j];
        }
        cout << endl;
}

// функція avrgcols() визначення середнього значення кожного стовпця
void  avrgcols(double matr[][maxcol], int rows, int cols)
{
        double sum, avr;
        for (int j = 0; j < cols; j++)
        {
                sum = 0.0;
                for (int i = 0; i < rows; i++)  sum += matr[i][j];
                avr = sum / rows;
                cout.precision(5);
        cout << "Average of  " << j << " column = " << avr << endl;
        }
}
//головна функція
int main()
{
        double matr[maxrow][maxcol];
        int rows, cols;
//введення кількості рядків і стовпців
        rows = getnum("rows ", minrow, maxrow);
        cols = getnum("columns ", mincol, maxcol);

inmatr(matr, rows, cols); //введення матриці
cout << endl;
avrgcols(matr, rows, cols); //обчислення середнього значення стовпців
return 0;
}
```

Результати обчислень:

```
Enter the number of rows  n from [2] to [30] : 2
Enter the number of columns  n from [1] to [20] : 4
Enter 0 matrix row
-2.34 4.89 0.41 4.97
Enter 1 matrix row
1.24 7.23 -5.56 7.16


Average of  0 column = -0.55
Average of  1 column = 6.06
Average of  2 column = -2.575
Average of  3 column = 6.065
```

Особливістю наведеної реалізації є те, що в пам’яті зберігається вся матриця ***matr[maxrow][maxcol]***, тому у функціях ***inmatr()*** та

avrgcols() перший параметр виглядає як *double matr[][maxcol]*. Це дає можливість обробляти «часткову» матрицю з розмірами, які вводить користувач у заданому діапазоні. Зрозуміло, що така побудова функцій є неоптимальною.

Треба нагадати, що ***за необхідності передачі багатовимірних масивів за допомогою покажчиків треба враховувати те, що в пам'яті такі масиви зберігаються як одновимірні***. Таким чином, при використанні, наприклад, матриці *a[n][m]* у пам'яті послідовно розміщуються *n* рядків по *m* елементів у кожному (рис. 6.4). Тому для роботи окремо з кожним рядком можна сформувати допоміжний масив покажчиків на ці рядки, тобто:

```
int a[n][m], *b[n];
for (i = 0; i<n; i++)
     b[i] = &a[i][0];
```

Тоді кожен елемент масиву *b[i]* зберігатиме адресу *i*-го рядка матриці *a[n][m]*, тому до кожного елемента *i*-го рядка *j*-го стовпця можна звертатися як *b[i][j]* або **(*(b+i)+j)*.

Використовуючи масив покажчиків **b[n]*, можна передавати як параметр функції подвійний покажчик (покажчик на покажчик), а у викликаючій функції достатньо вказати лише ім'я допоміжного масиву.

Приклад 9.6.

Для трьох заданих матриць переставити місцями парні та непарні рядки з використанням функції введення матриці, функції виведення матриці та функції перестановки.

```
#include<iostream>
using namespace std;

void inmatr(int**, int, int);    // прототип функції введення матриці
void outmatr(int**, int, int);   // прототип функції виведення матриці
void change(int**, int);         //прототип функції обміну рядків

int main()
{
        int a[3][4], * a_ptr[3];
        int b[4][5], * b_ptr[4];
        int c[6][6], * c_ptr[6];
        int i, j;

                // формування допоміжних масивів покажчиків
                for (i = 0; i < 3; i++) a_ptr[i] = &a[i][0];
                for (i = 0; i < 4; i++) b_ptr[i] = &b[i][0];
                for (i = 0; i < 6; i++) c_ptr[i] = &c[i][0];

        inmatr(a_ptr, 3, 4);     // виклик функції введення матриці
```

```
        cout << "\n Matrix a[3][4]:" << endl;
        outmatr(a_ptr, 3, 4);    //виклик функції виведення матриці
        change(a_ptr, 3); // виклик функції обміну рядків change()
        cout << "\nRezult matrix " << endl;
        outmatr(a_ptr, 3, 4);    //виклик функції виведення матриці
        // те ж саме з матрицями b[4][5] і c[5][5]
        inmatr(b_ptr, 4, 5);
        cout << "\n Matrix  b[4][5]:" << endl;
        outmatr(b_ptr, 4, 5);
        change(b_ptr, 4);
        cout << "\nRezult matrix " << endl;
        outmatr(b_ptr, 4, 5);
        inmatr(c_ptr, 6, 6);
        cout << "\n Matrix  c[6][6]:" << endl;
        outmatr(c_ptr, 6, 6);
        change(c_ptr, 6);
        cout << "\nRezult matrix " << endl;
        outmatr(c_ptr, 6, 6);
return 0;
}
//функція введення матриці inmatr()
void inmatr(int** x, int n, int m)
{
        for (int i = 0; i < n; i++)
                for (int j = 0; j < m; j++)
                 x[i][j] = rand()%100-50 ;
}
//функція обміну рядків change()
void change(int** x, int n)
{
        int* p;
        for (int i = 0; i < n - 1; i += 2)
        {
                p = x[i];
                x[i] = x[i + 1];
                x[i + 1] = p;
        }
}
//функція виведення матриці – outmatr()
void outmatr(int** x, int n, int m)
{
        for (int i = 0; i < n; i++)
        {
                cout << endl;
                for (int j = 0; j < m; j++)
                        cout << x[i][j] << "\t";
        }
        cout << endl;
}
```

Результати обчислень:

```
 Matrix a[3][4]:

-9      17      -16     -50
19      -26     28      8
12      14      -45     -5

Rezult matrix

19      -26     28      8
-9      17      -16     -50
12      14      -45     -5

 Matrix  b[4][5]:

31      -23     11      41      45
-8      -23     -14     41      -46
-48     3       42      32      -29
-34     -32     45      -3      -24

Rezult matrix

-8      -23     -14     41      -46
31      -23     11      41      45
-34     -32     45      -3      -24
-48     3       42      32      -29

 Matrix  c[6][6]:

21      -12     19      -38     17      49
-15     44      -47     -39     -28     -17
23      14      -9      -39     3       18
-3      -6      12      7       -13     9
-27     -9      -21     28      -34     -15
40      -8      38      -44     -10     -8

Rezult matrix

-15     44      -47     -39     -28     -17
21      -12     19      -38     17      49
-3      -6      12      7       -13     9
23      14      -9      -39     3       18
40      -8      38      -44     -10     -8
-27     -9      -21     28      -34     -15
```

У випадку динамічних масивів ситуація теж зрозуміла (ті ж самі функції, представлена тільки *main()*):

```
int main()
{
        int n,m,**a;
```

```
        int i, j;
        cout << "Input n = "; cin >> n;
        cout << "Input m = "; cin >> m;

        a = new int* [n];
        for (i = 0; i < n; i++)
                a[i] = new int[m];
        cout << "Matrix [" << n << "][" << m << "]:\n";

                inmatr(a, n, m);  // виклик функції введення матриці
                outmatr(a, n, m); //виклик функції виведення матриці
                change(a, n); //виклик функції обміну рядків change()
                cout << "\nRezult matrix " << endl;
                outmatr(a, n, m); //виклик функції виведення матриці

return 0;
}
```

Результат:

```
Input n = 6
Input m = 5
Matrix [6][5]:

-9      17      -16     -50     19
-26     28      8       12      14
-45     -5      31      -23     11
41      45      -8      -23     -14
41      -46     -48     3       42
32      -29     -34     -32     45

Rezult matrix

-26     28      8       12      14
-9      17      -16     -50     19
41      45      -8      -23     -14
-45     -5      31      -23     11
32      -29     -34     -32     45
41      -46     -48     3       42
```

9.3 Покажчики на функцію

Синтаксис мови C++ дозволяє використовувати покажчик на функцію. Кожна функція характеризується типом значення, що повертається, ім'ям і сигнатурою (списком типів параметрів функції). Під час використання імені функції без круглих дужок і параметрів ім`я функції може виступати як покажчик на цю функцію, а його значенням є адреса розміщення функції в пам'яті.

Ім'я будь-якої функції є **покажчиком-константою**, що дорівнює адресі початку входження у функцію, тобто адресі її першої машинної команди. Крім констант, можна також описувати **покажчики-змінні** на функцію у вигляді:

```
type (*name) (список аргументів); ,
```

де **type** — тип значення, що повертається функцією; ***name** — ім'я змінної покажчика на функцію.

Покажчики на функцію потрібні в таких випадках:

– для використання як формальних аргументів у інших функціях;

– для непрямого виклику інших (резидентних) функцій (програм), початок входу в які записується у відоме місце пам'яті.

Приклад 9.7.

Програмно реалізувати обчислення суми та різниці двох чисел з використанням покажчика на функцію для доступу до інших функцій.

```
#include<iostream>
using namespace std;

int difference(int, int);        // прототип функції difference()
int sum(int, int);               // прототип функції sum()

int main()
{
        int(*fun) (int, int);
        int x = 20, y = 5, z;
        cout << "x = 20, y = 5\n";
        // присвоювання покажчику fun адреси функції difference()
        fun = difference;
        z = fun(x, y);  //виклик функції fun()
        cout << "z = x - y = " << z << endl;
        // присвоювання покажчику fun адреси функції sum()
        fun = sum;
        z = fun(x, y);
        cout << "z = x + y = " << z << endl;
return 0;
}

// функція визначення різниці двох чисел – difference()
int difference (int a, int b)
{       return (a - b);   }

//   функція визначення суми двох чисел – sum()
int sum (int a, int b)
{       return (a + b);   }
```

Результати обчислень:

```
x = 20, y = 5
z = x - y = 15
z = x + y = 25
```

Покажчики на функції, як і звичайні змінні, можна об'єднати в масиви. Так, коли функції мають прототипи вигляду:

```
int f1(int, char*);
int f2(int, char*);
int f3(int, char*);
int f4(int, char*); ,
```

можна описати функцію

```
int(*fcmp[4]) (int, char*) { f1, f2, f3, f4 }; .
```

У результаті буде створено масив функцій, який матиме звичайний доступ до елементів, тобто:

```
int i = 0; fcmp[i](2, "  "); //виклик функції f1(2, "  ")
```

У цьому випадку, замінивши індекс, можна викликати іншу функцію.

9.4 Функції як параметр значень

Інколи доводиться у списку формальних аргументів (параметрів) функції використовувати інші функції. Така ситуація має місце, коли при звертанні до деякої функції треба викликати іншу функцію.

Параметр-функція записується у вигляді прототипу, тобто вказується тип функції, її ім'я і в дужках — перелік типів формальних аргументів або типів та імен формальних аргументів.

Приклад 9.8.

Скласти програму з використанням функції обчислення інтегралів методом трапецій (точність обчислення $e = 10^{-3}$)

$$Y = \int_{2,2}^{3} \sqrt{1 + \ln(x)}dx + \int_{0}^{1} \frac{\ln(1+x)}{(1+x)}dx.$$

```
#include<iostream>

using namespace std;
const  double E = 1.e-3;

//---------- підінтегральна функція 1-го інтеграла – fn1()
double fn1(double x)
{
```

```
        return sqrt(1 + log(x));
}

//-----------  підінтегральна функція 2-го інтеграла – fn2()
double fn2(double x)
{
        return  log(1 + x) / (1 + x);
}

//------------- функція методу трапецій – ft()
double ft(int n, double a, double b, double fun(double))
{// n - кількість точок розбиття відрізку
// a, b - початок та кінець відрізка
// fun() - підінтегральна функція
        int i;
        double s1, h, s = 0;
        do
        {
                s1 = s;
                h = (b - a) / n;
                s = (fun(a) + fun(b)) / 2;
                for (i = 1; i <= n - 1; i++)
                        s += fun(a + i * h);
                s *= h;       n *= 2;
        } while (fabs(s - s1) > E);
        return s;
}
// головна функція
int main()
{
        double y;
        y = ft(20, 2.2, 3.0, fn1) + ft(20, 0, 1.0, fn2);
        cout << "y = " << y << endl;
return 0;
}
```

Результат виконання програми:

```
y = 1.35746
```

При такому способі передачі функції як параметра значень можна використовувати як останній параметр будь-які функції, прототип яких відповідає *double ім'я_функції (double)*, це можуть бути функції з бібліотеки <*cmath*>. Наприклад, можливий виклик в *main():*

```
    y = ft(10, 1, 2, sin);
```

дає можливість розрахунку інтеграла на відрізку *[1;2]* з розбиттям на 10 точок функції *sin(x)* (результат 0.95625).

9.5 Перевантаження та шаблони функцій

Мова C++ надає можливість використовувати однакові імена для декількох функцій. Звичайно різні функції мають різні імена, але інколи виникає потреба у тому, щоб одна функція виконувала схожі дії над об'єктами різних типів. У цьому випадку є сенс визначити декілька функцій з однаковим іменем, але різним тілом. Такі функції повинні мати набори аргументів, які відрізняються, для того, щоб компілятор міг їх розпізнавати. В цьому випадку йде мова про **перевантаження функцій**. Головна перевага перевантажених функцій — це можливість визначання кількох функцій з однаковим іменем, але з різними типами або числом параметрів, при цьому тип результату, що повертається, може не змінюватися.

Приклад 9.9.

Написати приклад програмної реалізації з використанням перевантаження функцій.

```
#include<iostream>
#include<string.h>
using namespace std;
int funp (int x)                //---------  1 функція
{
        cout << "\n function int funp(int x)";
        return 3*x;
}

int funp (unsigned x)           //---------  2 функція
{
        cout << "\n function int funp(unsigned x)";
        return 2*x;
}
char funp (char x)              //---------  3 функція
{
        cout << "\n function char funp(char x)";
        return x + 3;
}

int funp (int x, const char* y) //---------  4 функція
{
        cout << "\n function int funp(int x, const char* y)";
        return x * strlen(y);
}

int funp (int x, char y)        //---------  5 функція
{
        cout << "\n function int funp(int x, char y)";
        return x * y;
```

```
}
float funp (float x)            //---------  6 функція
{
        cout << "\n function float funp(float x)";
        return x * x;
}
double funp (double x)          //---------  7 функція
{
        cout << "\n function double(double x)";
        return x + x;
}
int main()
{
        cout << "\nresult funp(5) = "<<funp(5) << endl;
cout << "\nresult funp((unsigned)5) = "<<funp((unsigned)5) << endl;
        cout << "\nresult funp('b') = " << funp('b') << endl;
cout << "\nresult funp (4, \"abc\") = " << funp (4, "abc") << endl;
        cout << "\nresult funp(4, 'f')  = " << funp(4, 'f') << endl;
cout << "\nresult funp((float)1.2)  = " << funp((float)1.2) << endl;
        cout << "\nresult funp(4.5)  = " << funp(4.5) << endl;
return 0;
}
```

Результат:

```
 function int funp(int x)
result funp(5) = 15

 function int funp(unsigned x)
result funp((unsigned)5) = 10

 function char funp(char x)
result funp('b') = e

 function int funp(int x, const char* y)
result funp (4, "abc") = 12

 function int funp(int x, char y)
result funp(4, 'f')  = 408

 function float funp(float x)
result funp((float)1.2)  = 1.44

 function double(double x)
result funp(4.5)  = 9
```

Процес пошуку відповідної функції із багатьох перевантажених полягає в знаходженні найкращого співвідношення типів формальних та фактичних параметрів. При виклику перевантаженої функції

компілятор визначає, яку саме функцію потрібно викликати, за типом фактичних параметрів. Тип, що повертається функцією, до уваги не приймається і в дозволі не приймає участі.

Дозволом перевантаження функції називається процес вибору тієї функції з множини перевантажених, яку треба викликати.

При вирішенні проблеми дозволу перевантаження функції виконуються такі кроки:

1. Виділяється множина перевантажених функцій для даного виклику, а також властивості списку аргументів, переданих функції.

2. Вибираються ті з перевантажених функцій, які можуть бути викликані з даними аргументами, з урахуванням їхньої кількості та типів.

3. Знаходиться функція, яка найкраще відповідає виклику.

Приклад 9.10.

З використанням перевантаження функцій написати програму обчислення площі квадрата, прямокутника та трикутника.

```
#include<iostream>
using namespace std;

int sfig(int a) // функція обчислення площі квадрата
{
        cout << "AREA of the square "; return a * a;
}

int sfig (int a, int b) // функція обчислення площі прямокутника
{
        cout << "AREA of the rectangle "; return a* b;
}

double sfig (int a, double h) // функція обчислення площі трикутника
{
        cout << "AREA of the triangle "; return a* h / 2;
}

int main()
{
        int a, b;
        double h;
//--------- введення сторони квадрата та обчислення площі

        cout << "\nEnter the side of the square: a = ";
        cin >> a;
        cout << "S = " << sfig(a) << "\n";
//--------  введення сторін прямокутника та обчислення площі

        cout << "\nEnter the sides of the rectangle: a = ";
```

```
        cin >> a;
        cout << "b = ";cin >> b;
        cout << "S = " << sfig(a, b) << "\n";
// введення основи та висоти трикутника та обчислення площі

        cout << "\nEnter the base and height of the triangle: a = ";
        cin >> a;
        cout << "h = "; cin >> h;
        cout << "S = " << sfig(a, h) << "\n";
return 0;
}
```

Результати:

```
Enter the side of the square: a = 5
AREA of the square S = 25

Enter the sides of the rectangle: a = 12
b = 4
AREA of the rectangle S = 48

Enter the base and height of the triangle: a = 7
h = 5.6
AREA of the triangle S = 19.6
```

У наведеній програмі перша функція відрізняється від інших кількістю параметрів, а друга та третя функції — типом параметрів та типом результату.

Використовуючи перевантажені функції, треба бути уважними з числовими аргументами-константами. Наприклад, якщо для обчислення площі трикутника викликати функцію таким чином:

```
    cout <<"s = "<< sfig(4, 5) << "\n";,
```

то отримаємо площу прямокутника, бо «5» — константа типу *int*, а потрібно *double*:

```
    cout<<"s = "<< sfig (4,5.0) << "\n";
```

або

```
    cout<<"s = "<< sfig (4,(double)5) << "\n";
```

Якщо дві перевантажені функції вимагають для встановлення відповідності стандартної трансформації фактичного аргументу, то виклик вважається неоднозначним і позначається компілятором як помилка. **Передбачається, що всі стандартні зміни вимагають одного обсягу роботи.** Наприклад, перетворення з *char* в *unsigned char* не більше пріоритетне, ніж з *char* в *double*.

Наприклад, якщо є дві перевантажені функції:

```
void fun (int) {…};
void fun (float) {…};
```

то виклик

```
fun (3.14)
```

буде неоднозначним: *fun (int)* не краще *fun (float)*, оскільки значення 3.14 типу *double*.

У процесі розв'язання багатьох задач необхідно використовувати функції, в яких алгоритм обчислення однаковий, а типи даних відрізняються. Прикладом є задачі пошуку та сортування. Особливістю програмування таких задач мовою C++ є використання шаблонів функцій.

Шаблони функцій — потужний засіб параметризації. За допомогою шаблона функції можна визначити алгоритм, який застосовуватиметься до даних різних типів, а конкретний тип даних передається функції у вигляді параметра на етапі компіляції.

Шаблон функції — це деяка узагальнена функція (родова функція) для сімейства функцій, призначених для розв'язання даної задачі. Визначається така шаблонна функція таким чином:

```
template <class Тип>
<тип_функції> <ім'я_функції> (<список_параметрів>)
{ // Тіло функції }
```

Тут параметр **Тип** — ім'я типу — задає тип даних, з яким працює функція. Цей параметр можна використовувати і всередині функції, однак при створенні конкретної версії узагальненої функції компілятор автоматично підставить замість нього фактичний тип. Традиційно узагальнений тип задається за допомогою ключового слова **class**, хоча замість нього можна застосовувати ключове слово **typename**.

Розглянемо дві функції:

```
int fmin (int a, int b) {return (a < b) ? a : b;}
double fmin (double a, double b) {return (a < b) ? a : b; }
```

Очевидно, що вони різняться тільки наявними типами *int* і *double*. Якщо ці типи позначити абстрактним типом, наприклад **Type**, можна створити шаблон функції. Виглядатиме це так:

```
template <class Type>   //оголосили назву абстрактного типу Type
Type fmin(Type a, Type b)
{
        return (a < b) ? a : b;
}
```

Поки немає виклику функції *fmin()*, при компіляції вона в бінарному коді не створюється. А якщо оголосити групу викликів функції зі змінними різних типів, то для кожного компілятор створить свою реалізацію на основі шаблону.

Виклик шаблонної функції, в загальному випадку, відповідає виклику звичайної функції. При цьому компілятор визначить, який тип використовувати замість *Type*, на підставі типу фактичних параметрів.

```
#include<iostream>
using namespace std;
template <class Type>
Type fmin(Type a, Type b)
{
        return (a < b) ? a : b;
}
int main()
{
        cout << fmin (1, 2) << endl;
        cout << fmin(3.1, 1.2) << endl;
return 0;
}
```

Результат:

```
1
1.2
```

У шаблоні функції може бути оголошено декілька формальних типів даних, а також використано параметри означених раніше типів. Наприклад:

```
template <class T1, class T2>
type_func my_func(T1 a, double x, T2 b, int c, char s)
{
        // оператори тіла функції
}
```

Таким чином, оголошення шаблонів функцій завжди починається з ключового слова *template* (шаблон), за ним у кутових дужках визначається список формальних типів, перед кожним з яких вказується ключове слово *class*. Далі йде звичайний опис функції. При цьому формальні типи, представлені у заголовку шаблону, можна використовувати в опису функції для задання типів аргументів функції, типу значення, що повертається, а також для оголошення змінних усередині тіла функції.

Приклад 9.11.

Написати шаблон функції, що повертає мінімальний елемент масиву, застосувати цю функцію для обробки масивів різних типів.

```
#include<iostream>
using namespace std;

template <class T> //----------- шаблон функції
T minmas(T* a, int k)
{
        T  min = a[0];
        for (int i = 1; i < k; i++)
                if (a[i] < min)   min = a[i];
        return  min;
}
// -------------------- головна функція
int main()
{
int b[10] = { 1, 6, 8, 5, 9, -6, 4, -5, 2, 7 };  // масив цілих чисел
        // виклик функції minmas() та виведення результатів
        cout << " min b[10] = " << minmas(b, 10);
        cout << endl;

float c[5] = { -4.5, 6.4, 7.0, -6.3, 2.1 };   // масив дійсних чисел

        cout << " min c[5] = " << minmas(c, 5);

return 0;
}
```

Результат обчислень:

```
 min b[10] = -6
 min c[5] = -6.3
```

У заголовку шаблону цієї функції оголошено єдиний формальний параметр *T* як тип даних, що повинні оброблятися функцією *minmas()*. У заголовку функції параметр *T* використовується для задання типу значення функції, що повертається *(T minmas)*,та для задання типу покажчика **a*. Усередині функції параметр *T* застосовано для визначення типу локальної змінної *min*. Завдяки цьому шаблону в програмі можна обробляти масиви різних типів.

9.6 Рекурсивні функції

Мова C++ підтримує можливість звернення функції самої до себе — **рекурсію**. Розрізняють пряму та непряму рекурсії. Функція називається **прямо рекурсивною**, якщо містить у своєму тілі виклик самої себе. Якщо ж функція викликає іншу функцію, що у свою чергу викликає першу, то така функція називається **непрямо рекурсивною**.

Рекурсивні функції найчастіше використовуються для компактної реалізації рекурсивних алгоритмів.

Класичні приклади використання рекурсії — реалізація операції піднесення до степені та обчислення факторіала числа. Зазначимо, що ці приклади популярні тільки через їхню зручність для пояснення поняття рекурсії, однак вони не дають виграшу в програмній реалізації порівняно з ітераційним способом розв'язання цих задач.

Розглянемо рекурсивну функцію обчислення факторіала. Для того, щоб отримати значення факторіала числа *n!*, необхідно помножити на *n* всі попередні натуральні числа. Тобто, можна записати:

$$n! = n \cdot (n-1)! = n \cdot (n-1) \cdot (n-2)! = \ldots = n \cdot (n-1) \cdot \ldots \cdot 1 \cdot 0!$$

Враховуючи, що **0!** = **1** та **1!** = **1**, наведемо приклад цієї функції:

```
long fact(long n)
{
   return (n > 1) ? n * fact(n - 1) : 1;
}
```

Рекурсивні функції є зручним засобом породження комбінаторних об'єктів заданого виду.

Приклад 9.12.

Вивести на екран усі зростаючі послідовності довжиною *k*, елементами яких є натуральні числа від 1 до *n*. Передбачається, що *k* не перевищує *n*, інакше таких послідовностей не існує.

Програма оперуватиме з масивом $a_0 \ldots a_{k-1}$ і цілою змінною *t*. Припускаючи, що $a_0 \ldots a_t$ — зростаюча послідовність натуральних чисел з відрізка *1...n*, рекурсивна функція *generate()* друкує всі її зростаючі продовження до довжини *k*.

```
#include<iostream>
using namespace std;

const int m = 20;
/* максимальна розмірність масиву зростаючої послідовності */

int t = 1, n, k, a[m];
/* k - довжина зростаючої послідовності
n - довжина відрізка натуральних чисел, з якого формуються
послідовності */

//рекурсивна функція генерації зростаючих послідовностей
void generate()
{
        int i;
        if (t == k) //одна з послідовностей повністю сформована
```

```
        {
                //виведення послідовності
                for (i = 0; i < k; i++)  cout << a[i] << ' ';
                cout << endl;
        }
        else    //формування послідовності
        {  t++;
                for (i = a[t - 2] + 1; i <= t - k + n; i++)
                {
                        a[t - 1] = i;
                        generate();
                }
                t--;
        }
}
int main()
{
        int j;
        cout << "input k, n (k<n)\n k = ";
        cin >> k;
        cout << "n = ";
        cin >> n;
        cout << endl;
        for (j = 1; j <= n - k + 1; j++)
        {
                a[t - 1] = j;
                generate();
        }
return 0;
}
```

Результат:

```
input k, n (k<n)
 k = 4
n = 6

1 2 3 4
1 2 3 5
1 2 3 6
1 2 4 5
1 2 4 6
1 2 5 6
1 3 4 5
1 3 4 6
1 3 5 6
1 4 5 6
2 3 4 5
2 3 4 6
2 3 5 6
2 4 5 6
3 4 5 6
```

Класичним прикладом використання рекурсії є також ***швидке сортування*** — один з найефективніших методів сортування. Швидке сортування (QuickSort) — це ефективний алгоритм сортування, заснований на принципі «розділяй та володарюй». Він рекурсивно розбиває масив на менші підмасиви, переміщуючи елементи, менші за опорний, ліворуч, а більші – праворуч, після чого рекурсивно сортує ці підмасиви.

Загальний алгоритм швидкого сортування здійснюється таким чином:

– з масиву вибирається деякий опорний елемент a_i (частіше — центральний);

– виконується процес розподілу масиву, що переміщує всі елементи, менші або рівні a_i, вліво від нього, а всі елементи, більші або рівні a_i, — вправо;

– тепер масив складається з двох підмасивів, причому лівий менше або дорівнює правому:

$\leq a_i$	a_i	$\geq a_i$

– для кожного з обох підмасивів, якщо в них більше двох елементів, знову знаходимо опорні елементи та виконуємо процес розділу, тобто здійснюємо рекурсію. Наприкінці отримаємо цілком відсортовану послідовність.

Розглянемо цей метод на прикладі.

Приклад 9.13.

Відсортувати одновимірний масив a_n за зростанням елементів, використовуючи рекурсивну функцію швидкого сортування.

У заданому масиві $a_0 \ldots a_{Nend}$ здійснимо розділення його елементів:

– виберемо з масиву центральний елемент ***c***, за яким здійснюватиметься розділ;

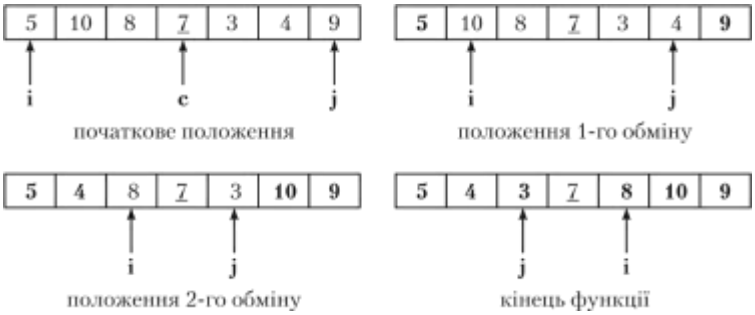
– введемо індекси ***i*** і ***j***, які на початку алгоритму вказують, відповідно, на лівий та правий елементи заданої послідовності;

– рухатимемося одиничним кроком за допомогою індексу ***i*** у напрямку до кінця масиву, поки не знайдемо елемент $a_i >= c$. Потім аналогічним чином почнемо рухатися за допомогою індексу j від кінця масиву до початку, доки не знайдемо $a_j = c$;

– далі, якщо $i = j$, міняємо елементи a_i і a_j місцями і продовжуємо рухати ***i*** та ***j*** за тими ж правилами;

– повторюємо попередній крок, доки $i = j$.

Роботу рекурсивної функції швидкого сортування для масиву $a_0 \ldots a_6$ *(QuickSortRecur())*, що має значення **5, 10, 8, 7, 3, 4, 9**, з центральним елементом $c = a_3$ можна подати як показано нижче.



Тепер масив розділений на дві частини: всі елементи лівої частини менше або дорівнюють ***c***, усі елементи правої — більше або дорівнюють ***c***. Розподіл завершено. Якщо підмасив, розташований ліворуч або праворуч від *c*, містить більше одного елемента, викликаємо рекурсивну функцію *QuickSortRecur()* для нього. Програмна реалізація матиме такий вигляд:

```
#include<iostream>
using namespace std;
template <class T>
void QuickSortRecur(T* a, int Nend)
{
// Початковий масив a[], a[Nend] – останній елемент
        int i = 0, j = Nend;
        T rab, c;
        c = a[Nend >> 1];
/* вибір центрального елемента з використанням операції побітового
зсуву вправо */
        do
        {
                while (a[i] < c) i++;
                while (a[j] > c) j--;
                if (i <= j)
                {       rab = a[i];
                        a[i++] = a[j];
                        a[j--] = rab;
                }
        }
        while (i <= j);

/* рекурсивні виклики QuickSortRecur(), якщо є, що сортувати */
        if (j > 0) QuickSortRecur(a, j);
        if (Nend > i) QuickSortRecur(a + i, Nend - i);
}

int main()
{
```

```
int a[] = {2, 7, 6, 9, 45, 4, 3, 67, 104, 1, 99, 72, 43, 8, 4, 28};

int n = sizeof(a) / sizeof(int);
        QuickSortRecur(a, n - 1);
        cout << "\n Result of quick array sorting" << endl;
        for (int i = 0; i < n; i++)
                cout << a[i] << ' ';
return 0;
}
```

Результати обчислень:

```
 Result of quick array sorting
1 2 3 4 4 6 7 8 9 28 43 45 67 72 99 104
```

Кількість кроків розподілу (глибина рекурсії) складає приблизно *ln(n)*, якщо масив поділяється на приблизно рівні частини. Загальна швидкодія цього методу сортування: *n·ln(n)*. Це один з найбільш швидкодіючих алгоритмів сортування. Часто використовується на практиці, особливо для великих обсягів даних. Також необхідно зауважити, що швидке сортування не є ефективним для коротких масивів.

Слід уникати типових помилок при використанні рекурсії. Пропуск базового значення чи некоректний запис кроку рекурсії може привести до проблем, оскільки процес не буде збігатися до базового значення, а це в свою чергу призводить до нескінченної рекурсії та істотних витрат пам'яті. Це може слугувати аналогом нескінченного циклу в ітеративному (нерекурсивному) процесі.

9.7 Запитання та завдання

1. Що таке функція, яка її структура?
2. Які існують способи передачі параметрів і повернення результату обчислень функції?
3. Наведіть приклади і особливості передачі параметрів у функцію за значенням, за посиланням і за покажчиком?
4. Що таке локальні та глобальні змінні?
5. Як використовувати покажчики на функцію?
6. Охарактеризуйте особливості використання масивів як аргументів функції.
7. Що таке перевантаження функцій?
8. Що таке шаблон функції? Наведіть приклади використання.
9. Охарактеризуйте рекурсивні функції.

Завдання.

Набуття практичних навичок програмної реалізації задач з функціями під час розв’язання завдань розділу вправ «Функції».

10 ФАЙЛИ

10.1 Поняття файлу

Під час розв'язання задач на комп'ютері часто виникає необхідність у використанні даних, які записані на зовнішніх носіях інформації (дисках) і оформлені у вигляді файлів даних [4 – 8, 12 – 14, 16, 18]. Незалежно від того, які дані містять файли (числа, символи, рядки, масиви, структури тощо), в мові С++ вони трактуються як потоки даних (stream), що є послідовністю байтів, які зчитуються або записуються.

За замовчуванням у кожній програмі С++ можна користуватися такими стандартними потоками: стандартного введення (**cin**), стандартного виведення (**cout**) та виведення помилок (**cerr**). Для того, щоб користуватися файлами, потоки повинні бути створені та закріплені за цими файлами. **Використання файлів даних у програмі передбачає виконання таких операцій**:

- створення потоку обміну даними між файлом і пам'яттю комп'ютера;
- зв'язування цього потоку з конкретним ім'ям файлу на диску та відкриття файлу;
- запис даних у файл або читання їх з файлу; закриття файлу.

Для реалізації цих операцій існують спеціальні класи, які містять конструктори створення необхідних потоків:

- **ifstream** — для створення потоку читання даних;
- **ofstream** — для створення потоку запису даних у файл;
- **fstream** — використовується як для запису даних у файл, так і їхнього читання.

Конструктори — це спеціальні функції, які мають таке саме ім'я, що й ім'я класу. Вони записуються як з параметрами, так і без параметрів. Конструктори з параметрами одночасно створюють відповідний потік, зв'язують його з файлом на диску, відкривають файл для роботи та мають такі форми запису:

```
        ofstream in ("iф", ios::out);
```

або

```
        ofstream ("iф");
        ifstream in ("iф", ios::in);
```

або

```
        ifstream (" ");
        fstream in ("iф", ios::in | ios::out);
```

де **in** — ім'я потоку, який створюється для роботи з файлом; **iф** — константа або змінна типу **char[ ]**, її значення — ім'я файлу на диску.

Перший з конструкторів використовується для запису даних у файл, другий — для читання даних з файла, а третій — як для запису, так і для читання даних, наприклад: *ofstream fout("myfile.dat", ios::out);*.

Цей запис створює потік з ім'ям *fout*, зв'язує його з файлом на диску, який має ім'я *myfile.dat* і відкриває цей файл для запису даних. Файл *myfile.dat* буде створено в тому ж каталозі, що і програма. Якщо треба створити файл в іншому місці, то для запису його імені треба вказати шлях, наприклад, *"c:\\pvp\\myfile.dat"*. Тепер цей файл буде записано на диску *c:* в каталозі *pvp*.

Зверніть увагу на те, що для запису шляху треба використовувати подвійні зворотні косі риски.

Можна також для роботи з файлами застосувати конструктори без параметрів:

```
ofstream in;
ifstream in;
fstream in; ,
```

де **in** — ім'я відповідного потоку, тоді для зв'язування потоку з ім'ям файлу на диску та відкриття його для роботи треба додатково використовувати функцію член відповідного класу, тобто:

```
in.open("iф", ios :: режим | ios :: режим); .
```

Наприклад, відкриття файлу для запису до нього даних матиме вигляд:

```
ofstream fout;
fout.open("c:\\pvp\\myfile.dat", ios::out);
```

Конструктори як з параметрами, так і без них, виконують однакову роботу, тому яким з них надати перевагу — справа користувача.

10.2 Використання файлів

Приклад 10.1. Створити файл на диску та записати до нього масив чисел. Прочитати цей файл і вивести його компоненти на екран.

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
        int i, mas[5];
        //запис елементів масиву в файл:
        ofstream fout("array.dat"); /* створення потоку fout і
відкриття файлу з іменем array.dat для запису */
```

```
        if (!fout) cout << "Cannot open file\n";
        for (i = 0; i < 5; i++)
        {
        cout << " Enter " << i << " element\n";
        cin >> mas[i]; //введення елемента масиву з клавіатури
        fout << mas[i] << " ";
        }    // запис елемента в файл
        fout.close();

        //--------- читання компонентів файлу та виведення на екран
        ifstream fin("array.dat"); // створення потоку fin для
                                   // читання файлу
        if (!fin) cout << " Cannot open file to reading\n";
        cout << "REZULT\n";
        for (i = 0; i < 5; i++)
        {
        fin >> mas[i]; // читання чергового елемента масиву із файлу
                cout << "mas[" << i << "]=" << mas[i] << " ";
        }
        cout << "\nFile reading\n";
        fin.close();
return 0;
}
```

Результат розв’язання прикладу має вигляд:

```
Enter  0 element
375
 Enter  1 element
630
 Enter  2 element
-378
 Enter  3 element
-238
 Enter  4 element
951
REZULT
mas[0]=375   mas[1]=630   mas[2]=-378   mas[3]=-238   mas[4]=951
File reading
```

Приклад 10.2.

Записати у файл матрицю ***matr[2][4]*** поелементно за рядками, прочитати її з файлу та вивести на екран.

```
#include<iostream>
#include<fstream>
using namespace std;

int main()
{
int matr[2][4], i, j;
```

```
        //запис матриці у файл
ofstream out("filemat", ios::out | ios::binary); //відкриття файлу
cout << "Input matrix 2 x 4:" << endl;

        for (i = 0; i < 2; i++)
        {
                for (j = 0; j < 4; j++)
                {
                cin >> matr[i][j]; // введення чергового елемента матриці
                out << matr[i][j] << " "; // запис у файл цього елемента
                }
        }

        out.close();

        //читання матриці з файлу
        ifstream in("filemat", ios::in | ios::binary); // відкриття файлу
        for (i = 0; i < 2; i++)
                for (j = 0; j < 4; j++)
                in >> matr[i][j]; // читання з файлу елемента матриці
        in.close();

        //виведення матриці на екран
        cout << "\nMatrix matr";
        for (i = 0; i < 2; i++)
        {
                cout << endl;
                for (j = 0; j < 4; j++)
                        cout << matr[i][j] << " ";
        }
return 0;
}
```

Результат розв’язання прикладу:

```
Input matrix 2 x 4:
5 7 1 8
1 3 2 1
Matrix matr
5 7 1 8
1 3 2 1
```

У цій програмі спочатку елементи матриці з клавіатури вводились у пам’ять комп’ютера, потім кожен з них записувався у файл з ім’ям ***filemat***. Для цього попередньо було створено потік ***out*** і відкрито файл на диску.

Потім було створено потік ***in*** для зчитування даних з файлу в пам’ять комп’ютера, тобто до матриці ***matr[i][j]***. Наприкінці програми матрицю виведено на екран.

Приклад 10.3.

Записати у файл 5 прізвищ, потім прочитати їх і вивести на екран.

```
#include<iostream>
#include<string.h>
#include<fstream>
using namespace std;
int main()
{
        char st[5][15];
        int i;

        //------------------------------ запис файлу
        ofstream fout("st_file.txt"); // відкриття файлу
        if (!fout) cout << "Cannot open file\n";
        for (i = 0; i < 5; i++)
        {
                cout << " Enter " << (i + 1) << " name\n";
                cin.getline(st[i], 15); // введення чергового імені
                fout << st[i] << endl; // запис імені в  файл
        }
        fout.close();
        // --------------------- читання файлу та виведення на екран
        cout << "\nReading file\n\n";
        ifstream fin("st_file.txt");
        if (!fin) cout << "Cannot open file.txt\n";
        for (i = 0; i < 5; i++)
        {
                fin.getline(st[i], 15);
                cout << st[i] << "  ";
        }
        fin.close();
return 0;
}
```

Результат розв’язання прикладу:

```
 Enter 1 name
Kostenko S.V.
 Enter 2 name
Usik O.S.
 Enter 3 name
Larina O.N.
 Enter 4 name
Neev B.T.
 Enter 5 name
Svetlenko A.S.

Reading file

Kostenko S.V.  Usik O.S.  Larina O.N.  Neev B.T.  Svetlenko A.S.
```

У попередніх програмах запис даних у файл та їхнє читання з файлу здійснювалось послідовно поелементно. Але записати або прочитати декілька даних (наприклад, масив чисел) можна однією операцією. Для цього використовують функції члени відповідних класів, які мають вигляд:

```
in.write((char*)&p, sizeof(p));   //для запису даних у файл
in.read((char*)&p,sizeof(p));     //для читання даних з файлу,
```

де **in** — ім'я потоку введення або виведення; **p** — змінна будь-якого типу, якщо змінна **p** має тип **char[ ]**, то операція її приведення не потрібна.

Приклад 10.4.

Записати у файл масив ***mas[]*** однією операцією, потім прочитати цей файл теж однією операцією в масив ***mas1[]*** і вивести цей масив на екран.

```
#include<iostream>
#include<fstream>
using namespace std;
const int n = 5;
int main()
{
int i, mas1[n], mas[n] = { 10, 20, 30, 40, 51 };

 ofstream fout("file_digits.dat");       // відкриття файлу для запису
   if (!fout)cout << "Cannot open file\n";
 fout.write((char*)&mas, sizeof(mas));  // запис масиву в файл
 fout.close();
 ifstream fin("file_digits.dat"); // відкриття файлу для читання
   if (!fin)cout << "Cannot open file for reading\n";
   fin.read((char*)&mas1, sizeof(mas1)); // читання масиву з файлу
        cout << "file reading\n";

        //------------- виведення масиву mas1[] на екран
        for (i = 0; i < n; i++)
                cout << mas1[i] << " ";
        cout << endl;
        fin.close();
return 0;
}
```

Результат роботи програми має вигляд:

```
file reading
10 20 30 40 51
```

Розглянемо, наприклад, створення файлу, в якому необхідно записати список прізвищ абонентів та їхні телефони (дані типу

структура), а потім за потреби виведення на екран або всього списку, або тільки потрібних прізвищ і відповідних номерів телефонів.

Приклад 10.5.

Розробити програму, за допомогою якої здійснюється запис даних типу структура (список прізвищ абонентів та їхніх телефонів) у файл з ім'ям *struct.dat.*

```
#include<iostream>
#include<string.h>
#include<fstream>
using namespace std;

struct telefon
{
        char name[15];
        char tel[15];
};

int main()
{
        int i;
        telefon list[5];
        ofstream out("struct.txt");
        if (!out) cout << "Cannot open file\n";
        for (i = 0; i < 5; i++)
        {
        cout << "Enter " << i + 1 << " name and tel_number\n";
        //---------------------- введення імені та телефона з клавіатури
                cin >> list [i].name;
                cin >> list [i].tel;
        //---------------------- запис імені та телефона в файл
                out << list [i].name << "  " << list [i].tel << endl;
        }
        out.close();
return 0;
}
```

Результат роботи цієї програми такий:

```
Enter 1 name and tel_number
Ganna 0992345634
Enter 2 name and tel_number
Alex 0971234589
Enter 3 name and tel_number
David 0345678923
Enter 4 name and tel_number
Kelly 4447124523
Enter 5 name and tel_number
Lion 4823895672
```

Приклад 10.6.

Розробити програму читання файлу (*struct.dat*), створеного у прикладі 10.5, і виведення на екран за запитом користувача або списку прізвищ абонентів і їхніх телефонів, або тільки прізвища і номера телефону потрібного абонента.

```
#include<iostream>
#include<fstream>
using namespace std;
struct telefon
{
        char name[15];
        char tel[15];
};
int main()
{
        int i, p; char name1[15];
        bool t;
        telefon list;
//------------------- відкриття файлу, що створений раніше
        ifstream in("struct.txt");
        if (!in) cout << "\nCannot open file to reading\n";
//------------------- вибір режиму роботи з файлом
        cout << "What to do: reading list(1) or name(2)\n";
        cin >> p;   //-------- введення 1-го або 2-го режиму
        if (p == 1) //-------обробка 1-го режиму
        {
while (in >> list.name >> list.tel) // читання даних з файлу
                                            // «struct.dat»
cout << list.name << " " << list.tel << endl; // виведення даних
                                                    // на екран
        }
        else if (p == 2) //------------- обробка 2-го режиму
        {
                t = true;
                cout << "Enter name\n";
                cin >> name1; //-----  введення імені
                //цикл для читання даних з файлу
                while (in >> list.name >> list.tel)
                        if (strcmp(list.name, name1) == 0)
                        {
cout << list.name << " " << list.tel << "\n"; //виведення імені
                                                    //та телефона
        t = false;
                        }
                in.close();
        if (t) cout << "Name " << name1 << " missing\n";
        }
return 0;
}
```

Результат розв'язання програми для першого режиму її роботи, коли потрібно вивести усі дані з файлу:

```
What to do: reading list(1) or name(2)
1
Ganna 0992345634
Alex 0971234589
David 0345678923
Kelly 4447124523
Lion 4823895672
```

Результат розв'язання цієї програми для другого режиму роботи, коли необхідно вивести задане прізвище та телефон, має вигляд:

```
What to do: reading list(1) or name(2)
2
Enter name
Lion
Lion 4823895672
```

Функції *write()* і *read()* зручно використовувати для організації прямого доступу до даних у файлі. Але для цього потрібні також функції члени, які дозволяють переміщати покажчик потоку в будь-яке місце файлу. Ці функції застосовуються під час запису даних у файл та під час їхнього читання з файлу та мають відповідно вигляд:

```
in.seekp(n, dir);
in.seekg(n, dir); ,
```

де **in** — ім'я відповідного потоку (введення або виведення);

n — параметр, що вказує кількість байт, на яку треба перемістити покажчик потоку;

dir — необов'язковий параметр, що вказує на спосіб переміщення покажчика і приймає одне зі значень:

ios::beg — переміщення від початку файлу; **ios::cur** — переміщення від поточної позиції; **ios::end** — переміщення від кінця файлу.

Якщо параметр *dir* відсутній, то переміщення покажчика здійснюється з початку файлу.

Розглянемо програму, в якій використовуються методи прямого доступу до даних у файлі.

Приклад 10.7.

Записати у файл числову матрицю розміром 3 × 5, потім, користуючись засобами прямого доступу, прочитати з файлу спочатку другий рядок, за ним перший, а потім третій.

```
#include<iostream>
#include<fstream>
#include<iomanip>
```

```
using namespace std;

const int n = 3, m = 5;

int main()
{
        int i, j, mas[m];
        char fname[] = "matr.dat";
        ofstream fout(fname);
        if (!fout) cout << "Cannot open file\n";
        // -------------- введення та запис у файл матриці по рядках
        for (i = 0; i < n; i++)
        {
                cout << "Enter " << i << " matrix row\n";
                for (j = 0; j < m; j++)
                        cin >> mas[j];
                fout.write((char*)&mas, sizeof(mas));
        }

        fout.close();
        ifstream fin(fname);
        if (!fin) cout << "Cannot open file for reading\n";
        cout << "\nOutput of 2, 1 and 3 rows of the matrix\n\n";

        //------------ читання з файлу та виведення 2 рядка:
        fin.seekg(1 * sizeof(mas));
        fin.read((char*)&mas, sizeof(mas));
        for (j = 0; j < m; j++)
                cout << setw(3) << mas[j];
        cout << endl;
        //------------ читання з файлу та виведення 1 рядка:
        fin.seekg(0);
        fin.read((char*)&mas, sizeof(mas));
        for (j = 0; j < m; j++)
                cout << setw(3) << mas[j];
        cout << endl;
        //------------ читання з файлу та виведення 3 рядка:
        fin.seekg(2 * sizeof(mas));
        fin.read((char*)&mas, sizeof(mas));
        for (j = 0; j < m; j++)
                cout << setw(3) << mas[j];
        cout << endl;
        fin.close();
return 0;
}
```

Результат роботи програми:

```
Enter 0 matrix row
1 1 1 1 1
Enter 1 matrix row
```

```
2 2 2 2 2
Enter 2 matrix row
3 3 3 3 3

Output of 2, 1 and 3 rows of the matrix

  2  2  2  2  2
  1  1  1  1  1
  3  3  3  3  3
```

10.3 Запитання та завдання

1. Що таке файл, чим він відрізняється від масиву?
2. Які операції треба виконувати під час роботи з файлом даних?
3. Які існують засоби створення потоків і відкриття файлу?
4. Які дані можна записувати у файл?
5. Які функції можна використовувати для організації прямого доступу до даних у файлі?
6. Охарактеризуйте на прикладах порядок створення файлу.

Завдання.

Набуття практичних навичок роботи з файлами під час розв’язання завдань розділу вправ «Файли».

11 ДИРЕКТИВИ ПРЕПРОЦЕСОРА

В інтегроване середовище підготовки програм мовою С++ входить препроцесор. Призначення препроцесора – обробка вхідного тексту програми **до її компіляції**.

Перш ніж приступити до опису препроцесора, згадаємо його історію. Препроцесор мови С ++ успадкований від мови С. Більш того, він практично збігається з препроцесором мови С. Єдина відмінність між ними полягає в степені їхнього важливості.

Для керування препроцесором використовуються директиви препроцесора, кожна з яких розміщується в окремому рядку та починається символом «**#**».

У мові С кожна директива препроцесора є необхідною.

У мові С++ деякі надлишкові директиви компенсуються елементами самої мови. Фактично однією з довгострокових цілей мови С++ є повне виключення препроцесора. Проте на сьогодні директиви препроцесора застосовуються дуже часто, та в досяжному майбутньому це положення не зміниться.

Препроцесор містить наступні директиви:

#define	**#elif**	**#else**	**#endif**
#error	**#if**	**#ifdef**	**#ifndef**
#include	**#line**	**#pragma**	**#undef**

11.1 Директива #define

Директива **#define** визначає підстановку в тексті програми. Вона використовується для визначення:

– символічних констант (всі входження імені замінюються на текст підстановки):

```
#define ім'я текст_підстановки
```

– ***макросів,*** які виглядають як функції, але реалізуються підстановкою їхнього тексту в текст програми:

```
#define  ім'я (параметри) текст_підстановки
```

– символів, керуючих умовної компіляцією. Вони використовуються разом з директивами #ifdef і #ifndef. Формат:

```
#define ім'я
```

Приклади:

```
#define VERSION 1
#define VASIA “Василь Іванович”
#define MAX (x,y) ((x)>(y)?(x):(y))
#define MUX
```

Імена рекомендується записувати великими літерами, щоб візуально відрізняти їх від імен змінних і функцій. Параметри макросу використовуються при макропідстановці. Наприклад, якщо в тексті програми використовується виклик наведеного вище макросу у вигляді

```
y= MAX (sum1, sum2);
```

він буде замінений на

```
y =((sum1)>(sum2)?(sum1):(sum2));
```

Відсутність круглих дужок може призвести до неправильного порядку обчислення, оскільки препроцесор не оцінює текст, який вставляється, з точки зору синтаксису. Наприклад, якщо до макросу

```
#define sqr (x) (x * x)
```

звертатися як *sqr (y + l)*, в результаті підстановки вийде вираз *(y + 1 * y + 1)*.

Макроси та символічні константи успадковані з мови С, під час написання програм на С++ їх треба уникати. Замість символічних констант краще використовувати const або enum, а замість макросів – вбудовані функції або шаблони.

Приклад програми, де показана робота директиви **#define**:

```
#include<iostream>

using namespace std;
#define cmp >
#define sort(mas, n)\
{                    \
for (int j = 0; j < n; j++)\
        for (int i = 0; i < n - 1 - j; i++)\
                if (mas[i] cmp mas[i + 1])   \
                {                           \
                        mas[i] = mas[i] + mas[i + 1]; \
                        mas[i + 1] = mas[i] - mas[i + 1]; \
                        mas[i] = mas[i] - mas[i + 1];   \
                }                            \
                        \
}
#define PRINT(mas, n)\
for (int i = 0; i < n; i++)\
        cout << vect[i] << " "; \
        cout << endl; \
```

```
        int main()
{
#define n 5
                int vect[n] = { 12,3,1,6,8 };
                cout << "source array:\n";
                PRINT(vect, n);
                sort(vect, n);
                cout << "sorted array:\n";
                PRINT(vect, n);
return 0;
}
```

Результат:

```
source array:
12 3 1 6 8
sorted array:
1 3 6 8 12
```

11.2 Директива умовної компіляції

Директиви умовної компіляції **#if**, **#ifdef** і **#ifndef** застосовуються для того, щоб виключити компіляцію окремих частин програми. Це буває корисно при налагодженні або, наприклад, за підтримки декількох версій програми для різних платформ.

Формат директив **#if**:

```
#if константний_вираз
[#elif константний_вираз
[#elif константний_вираз
...]
[#e1se
...]
#endif
```

Кількість директив **#elif** – довільна. Блоки, що виключаються до коду, можуть містити як опис, так і виконувані оператори. Приклад умовного включення різних версій заголовного файлу:

```
#if VERSION == 1
#define INCFILE "vers1.h"
#elif VERSION == 2
#define INCFILE "vers2.h" /* і так далі */
#else
#define INCFILE "versN.h"
#endif
#include INCFILE
```

Інше призначення директиви – тимчасово закоментувати фрагменти коду, наприклад:

```
#if 0
int i, j;
double x, y;
#endif
```

Оскільки допускається вкладеність директив, такий спосіб досить зручний.

Найбільш часто в програмах використовуються директиви **#ifdef** і **#ifndef**, що дозволяють управляти компіляцією залежно від того, чи визначений за допомогою директиви #define зазначений у них символ (хоча б як порожній рядок, наприклад, #define 32_BIT_SUPP0RT):

```
#ifdef символ
// Розташований нижче код компілюється, якщо символ визначений
#ifndef символ
// Розташований нижче код компілюється, якщо символ не визначений
```

Дія цих директив поширюється до першого *#elif*, *#else* або *#endif*.

Директива *#ifndef* часто застосовується для того, щоб забезпечити включення заголовного файлу тільки один раз.

11.3 Директиви #ifdef та #ifndef

Інший спосіб умовної компіляції заснований на застосуванні директив **#ifdef, #ifndef,** що означають "*if defined*" (якщо визначено) та "*if not defined*" (якщо невизначено). Загальний вид директиви **#ifdef** такий:

```
#ifdef ім'я_макросу
послідовність операторів
#endif
```

Наприклад:

```
// Перевірка на повторне включення файлу
#ifndef Unit2H
#define Unit2H
// Тіло заголовного файлу
#endif
```

Якщо ім'я макросу раніше визначено за допомогою директиви **#define,** компілюється відповідний блок коду.

Директива **#ifndef** має наступний загальний вигляд:

```
#ifndef ім`я_макросу
послідовність операторів
#endif
```

Якщо ім'я макросу не було визначено за допомогою директиви **#define,** компілюється відповідний блок коду.

Директиви **#ifdef** і **#ifndef** можуть використовувати директиви **#else** і **#elif**.

Наприклад, наступний фрагмент програми виводить на екран рядок "*Hello, Tim*" і "*Rem went to training*". Однак, якщо іменована константа *TED* не визначена, на екран виводиться повідомлення "*Hello everyone*", а потім – "*Rem went to training*".

```
#include<iostream>
using namespace std;
#define TED 10
int main()
{
#ifdef TED
        cout<<"Hello, Tim\n";
#else
        cout << "Hello everyone\n";
#endif
#ifndef RALPH
        cout << "Rem went to training\n";
#endif
return 0;
}
```

Директиви **#ifdef** і **#ifndef** можуть бути вкладеними, причому стандарт мови С допускає як мінімум вісім рівнів вкладення, а стандарт мови С++ – до 256.

```
#ifndef HEADER_INCLUDED
#include "myheader.h”
#define HEADER_INCIUDED
#endif
```

Директива #undef

Директива **#undef ім'я** видаляє визначення символу. Використовується рідко, наприклад, для відключення якої-небудь опції компілятора.

Оператор defined

Крім директиви **#ifdef,** є ще один спосіб перевірити, чи визначено ім'я макросу. Для цього можна застосовувати директиву **#if** в поєднанні зі статичним оператором **defined**, виконуваним на етапі компіляції. Оператор **defined** має такий загальний вигляд:

```
defined ім`я_макросу
```

Якщо в даний момент ім'я макросу є визначеним, вираз має значення «істина». В іншому випадку він помилковий. Наприклад, щоб розпізнати, чи є іменована константа **MYFILE** визначеною, можна використовувати одну з таких команд препроцесора:

```
#if defined MYFILE
```

чи

```
#ifdef MYFILE
```

Поставивши перед оператором **defined** символ **!**, можна задати протилежне умові значення. Наприклад, наступний фрагмент програми компілюється тільки в тому випадку, якщо іменована константа **DEBUG** не визначена.

```
#if !defined DEBUG
printf("Final version!\n");
#endif
```

Одна з причин, по якій використовується оператор **defined,** полягає в тому, що він дозволяє перевірити, чи визначено ім'я макросу за допомогою оператора **#elif.**

Імена зумовлених макросів

У мові C ++ існує шість вбудованих імен макросів:

```
__LINE__
__FILE__
__DATE__
__TIME__
__STDC__
__cplusplus__
```

Макроси **__LINE__ і __FILE__** містять номер поточного рядка та ім'я файлу, який компілюється.

Макрос **__DATE__** містить рядок, що має вигляд місяць/день/рік. В нього записується дата компіляції файлу.

Макрос **__TIME__** містить час компіляції програми, представлений у форматі *години: хвилини: секунди.*

Сенс макросу **__STDC__** залежить від реалізації. Як правило, якщо макрос **STDC__** визначений, компілятор суворо дотримується стандарту мови C і C ++, не допускаючи жодних нестандартних розширень.

Компілятор, що суворо підтримує стандарт мови C++, визначає макрос __cplusplus, значення якого містить як мінімум шість цифр. Компілятори, які дозволяють нестандартні розширення, визначають

значення макросу cplusplus, що складається з п'яти та меншої кількості знаків.

11.4 Директиви #line, #error, #pragma

Директива **#line** відповідно змінює вміст макросів __LINE__ і **__FILE__,** що становлять ідентифікатори, визначені компілятором. Ідентифікатор **__LINE__** містить номер поточного рядка коду, який компілюється. Ідентифікатор **__FILE__** є рядком, що містить ім'я файлу, який компілюється. Загальний вигляд директиви **#line** такий:

```
#line номер "ім'я файлу"
```

Тут параметр *номер* дорівнює кожному позитивному числу, яке стає новим значенням макросу __LINE__. Параметр *ім'я файлу* не обов'язковий і може бути будь-яким допустимим ім'ям файлу, яке стає новим значенням макросу **__FILE__.** Директива **#line** в основному використовується при налагодженні програм і в спеціальних додатках.

Наприклад, наступний код встановлює, що нумерацію рядків треба починати з числа 100. Функція *printf ()* виводить на екран число 102, оскільки вона знаходиться в третьому рядку програми, відраховуючи від директиви *#line 100***.**

```
#include<iostream>
#line 100
using namespace std;
int main()  //101
{           //102
        cout << __LINE__ << endl; //103
        cout << __FILE__ << endl; //Повне ім`я файлу
return 0;
}
```

Результат:

```
103
C:\Users\XE\source\repos\aaa\aaa.cpp
```

Директива # error

Директива **#error** змушує компілятор припинити компіляцію. Вона використовується в основному при налагодженні програм. Загальний вигляд директиви **#error** такий.

```
#error повідомлення_про_помилку
```

Параметр *повідомлення_про_помилку* є рядком, що не укладений в лапки. Під час виконання директиви **#error** на екран виводиться

повідомлення про помилку, яке може супроводжуватися іншою інформацією, визначеною компілятором.

Директива # pragma

Директива **#pragma** залежить від реалізації. Вона дозволяє передавати компілятору різні команди. Наприклад, компілятор може мати опцію, яка передбачає трасування виконання програми. Цю опцію можна задати за допомогою директиви **#pragma.** Деталі і допустимі опції перераховуються в документації, яка супроводжується компілятором.

11.5 Оператори препроцессора # та

Оператор **#**, який називається **оператором перетворення в рядок** (*stringize*), перетворює свій аргумент в рядок, укладену в лапки. Як приклад розглянемо наступну програму.

```
#include<iostream>
using namespace std;
#define mkstr(s) cout<<#s
int main()
{
        mkstr (this is an example macros);
return 0;
}
```

Препроцесор перетворить рядок *mkstr (this is an example macros);* в рядок *cout<<"this is an example macros";.*

Оператор **##**, званий оператором **конкатенації** (*pasting*), "склеює" дві лексеми. В наступній програмі демонструється робота директив препроцесора.

```
#include<iostream>
#include<cmath>
using namespace std;
#define MKSTR(s) cout<<#s
#define ABC(a,b,c,d) cout<<a(b##c##d)
#define PASS cout<<endl
int main()
{       MKSTR (this is an example macros);
        PASS;
        double x = 0.25, y = 0.47;
        MKSTR(x = )<<x; MKSTR(\t); MKSTR(y = )<<y;
        PASS;
        MKSTR(sin(x + y) = );
        ABC(sin, x, +, y);
        PASS;
        MKSTR(log(x / y) = );
```

```
        ABC(log, x, /, y);
return 0;
}
```

Результат:

```
this is an example macros
x =0.25 y =0.47
sin(x + y) =0.659385
log(x / y) =-0.631272
```

11.6 Запитання та завдання

1. Назвіть стадії препроцесорної обробки.
2. Що таке макрос?
3. Навіщо в макросі використовуються дужки?
4. Чим відрізняється макрос від вбудованих функцій?
5. Чим відрізняється макрос від шаблонних функцій?
6. В чому сенс умовної компіляції?

12 ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

12.1 Основні визначення об'єктно-орієнтованого програмування (ООП)

Основні ідеї ООП були закладені ще у 1960-х роках і були реалізовані в першій об'єктно-орієнтованій мові Simula/Simula67. Головна парадигма ООП в тому, що головний елемент програми це об'єкт, який розглядається як «чорний ящик», яким можливо керувати через інтерфейс. Наприклад, як ми керуємо телевізором через пульт дистанційного керування, чи автомобілем, застосовуючи кермо, педалі та коробку переключення передач. Пов'язуючи між собою «незалежні»/»самодостатні» об'єкти програміст створює програму. Таким чином програма в рамках ООП, це множина пов'язаних між собою об'єктів. При створенні об'єкта виділяється 3 основні концепції: інкапсуляція, успадкування, поліморфізм (пізніше в ООП було додано четверту концепцію «абстракція», проте в класичному C++ це реалізовано як частина поліморфізму як абстрактні класи).

Розглянемо основні визначення, зауважимо, що не існує загального єдиного тлумачення і в літературі наведено безліч інтерпретацій. В більшості об'єктно-орієнтованих мов виділяють наряду з поняттям об'єкт, ще поняття клас, де клас описує тип, а об'єкт змінна або реалізація від цього класу. Наприклад, якщо розглянути «проєкт будинку» і побудовані «будинки», то «проєкт будинку» – це клас, а побудовані «будинки» – це об'єкти від цього класу.

Розглянемо основні постулати ООП і яким чином вони реалізовані в C++ (примітка: в інших мовах програмування ООП може бути реалізовано по-іншому).

Інкапсуляція – це об'єднання даних з кодом, який обробляє ці дані, в один об'єкт та захист даних від неправильного застосування. Це концепція реалізації «чорного ящика», приховування деталей реалізації класу. Зазвичай невідомо з яких даних він складається, проте відомо, як його застосовувати. Наприклад, ми не знаємо як реалізований калькулятор у середині, проте знаємо, як отримати значення деякого виразу для вхідних параметрів.

Інкапсуляція в C++ реалізована через клас, захист даних реалізований через розташування полів класу у розділах *private/protected*. При розташуванні полів класу у розділі *public* говорять про порушені інкапсуляції у класі.

Успадкування чи наслідування – це можливість створення нового об'єкта на базі існуючого з можливістю розширення або заміни

його властивостей. Такий об'єкт називають **нащадком** або **похідним**. Об'єкт, на основі якого створений нащадок, називають *батьківським* або *базовим*.

Успадкування в C++ реалізоване через успадкування класів.

Поліморфізм (від грецького polymorphos) – це властивість, яка дозволяє одне і те ж ім'я використовувати для вирішення двох або більше схожих, але технічно різних завдань. Метою поліморфізму, стосовно об'єктно-орієнтованого програмування, є використання одного імені для завдання загальних для класу дій. У більш загальному сенсі, концепцією поліморфізму є ідея «один інтерфейс, безліч реалізацій методів».

Це означає, що можна створити загальний інтерфейс для групи близьких за змістом дій.

Наприклад, навчившись керувати одним легковим автомобілем (тобто Ви освоїли інтерфейс автомобіля), Ви можете керувати також і іншими легковими автомобілями. І Вас «не цікавить», що, наприклад, коробка передач чи гальма в різних автомобілях технічно реалізовані по-різному, Ви просто їх використовуєте.

Поліморфізм в C++ реалізований через перевизначення методів базового класу у класах нащадках. На відміну від інших мов в C++ також виділяють 2 типи поліморфізму: *динамічний* та *статичний*. Перший реалізується через віртуальні методи, другий через звичайні (не віртуальні) методи.

Надалі під **інтерфейсом** класу матимемо на увазі його методи, і, якщо не вказано інше, матимемо на увазі усі відкриті (*public*) методи.

Примітка

Існує думка, що перевантаження методів також може належати до поліморфізму, проте розробники цього підручника схиляються до думки, що під визначенням «одне і те ж ім'я» ми маємо на увазі однакове ім'я методу плюс однакові параметри цього методу, плюс константний чи не константний метод. Отже, якщо відштовхуватися від такого тлумачення визначення «одне і те ж ім'я», то до поліморфізму належить тільки перевизначення методу у класі нащадку.

12.2 Синтаксис опису простого класу

Клас є типом даних, що визначається користувачем і є моделлю реального об'єкта у вигляді даних та методів для роботи з ними. Іншими словами клас – це об'єднання даних з кодом для обробки цих даних. **Клас** – це інструмент у мові C++ для реалізації інкапсуляції. **Дані** – це змінні, які описані у класі. Їх ми називатимемо **полями**

класу. **Методи** – це функції, які описані у класі. **Об’єкт –** це змінна типу *класу*.

Примітка
У літературі може зустрічатися також інша термінологія – дані-члени (поля) і функції-члени (методи) класу.

Говорять, що поля задають стан об’єкта, а методи – його поведінку.

Оголошення простого класу без наслідування має таку форму:

```
class [ім’я класу]
{
    [public:] // доступно всім
    [дані, методи, властивості, події]
    [protected:]// доступний тільки нащадкам
    [дані, методи, властивості, події]
    [private:] // доступні тільки у класі
    [дані, методи, властивості, події]
} [список об’єктів];
```

Зауваження:

- те, що взято в дужки «*[]*», означає «необов’язково» і може бути відсутнім. Тобто наступний пустий клас хоч і не має сенсу, проте правильний з точки зору синтаксису*:*

```
class {};
```

- якщо розділ *(public* | *protected* | *private)* не вказаний, то за замовчуванням застосовується *private;*
- **//** – означає коментар, те, що ігнорується компілятором.

Приклад 12.1.

Розглянемо створення «простого» класу "Дата". Клас зберігає день, місяць, рік. Здатний встановити дату, повідомити її. (Що таке методи, поля та інші елементи класу детальніше буде розглянуто після прикладу).

```
class Tdate
{
public:
        int day, month, year;
        void set(int, int, int);
};
```

Визначення методу можна розташувати всередині або за межами оголошення класу. У першому випадку транслятор спробує створити метод *inline*. У другому – перед ім’ям методу, повинно передувати ім’я класу разом з операцією доступу **::** .

Всі дані класу доступні з методів того ж класу.

Визначення методу (функції-члена) *TDate::set*, розташоване за межами оголошення класу.

```
void Tdate::set(int d, int m, int y) {
        day = d;
        month = m;
        year = y;
}
```

Клас є типом даних. Самі ж дані такого типу називають **екземплярами класу** або **об’єктами**. Кожен об’єкт має власну копію всіх даних класу. Щоб створити об’єкт у пам’яті, його треба визначити, наприклад:

```
    Tdate date;
```

Після створення об’єкта отримати доступ до його даних і методів можна за допомогою операції прямого вибору, що позначається крапкою, точно так само, як і до полів структури.

```
int main()
{ Tdate date;

    date.set(10, 12, 2017);
    cout << date.day <<"/"<< date.month <<"/"<< date.year << endl;
    system("pause");

return 0;     }
```

У цьому прикладі порушено принцип інкапсуляції, а саме – відкритий прямий доступ до даних класу (*day, month, year*). На практиці цього уникають.

Захистимо дані від явного втручання, перемістивши їх у розділ *private*. Для друку даних класу реалізуємо метод *print()*.

Дані класу *Tdate* приховані від користувача, а методи відкриті. Далі наведений повний приклад класу з його застосуванням.

```
#include<iostream>
#include<stdlib.h>
using namespace std;
class Tdate
{
    private:
        int day, month, year;
    public:
        void set(int, int, int);
        void print() const;
};

void Tdate::set(int d, int m, int y)
```

```
{
        day = d;
        month = m;
        year = y;
}

void Tdate::print() const
{
        cout << day <<"/"<< month <<"/"<< year << endl;
}

int main()
{
        Tdate date;
        date.set(10, 12, 2025);
        date.print();
        system("pause");
return 0;
}
```

Результат роботи програми:

```
10/12/2025
```

12.3 Специфікатори доступу

Класи в C++ мають три різних рівні доступу до своїх елементів, тобто полів та методів:

- закриті елементи (private);
- захищені елементи (protected);
- відкриті елементи (public).

До даних у ***закритому розділі*** (**private**) мають доступ тільки методи свого класу. Класам-нащадкам (детальніше у підрозділі 12.14) забороняється доступ до закритих даних своїх базових (батьківських) класів. За замовчуванням усі елементі класу мають атрибут **private** (закритий).

До даних у ***захищеному розділі*** (**protected**) мають доступ методи свого класу та методи класів-нащадків.

У свою чергу, до даних ***відкритого розділу*** (**public**) можуть звертатися будь-які методи чи зовнішні функції.

Існують такі правила створення розділів класу:

- розділи можуть з’являтися в будь-якому порядку і декілька разів;
- якщо не оголошено жодного розділу, компілятор за замовчуванням оголошує усі елементи закритими;
- розміщати дані елементи класу у відкритому розділі не рекомендується. Дані елементи класу звичайно розміщують

у закритому, або захищеному розділі, щоб до них мали доступ методи класу, а також методи нащадків;

– для зміни значень даних (полів) необхідно використовувати методи класу, які зазвичай називають «геттерами» (для одержання даних) і «сеттерами» (для зміни даних).

Наприклад:

```
class Demo
{
        int a;
    public:
        int b;
    protected:
        int c;
    private:
        int d;
   public:
        void set() { a = 1; b = 2; c = 3; d = 4; }
//правильно, метод set() має доступ до усіх розділів власного класу
};
//Створимо клас нащадок, детальніше у розділі 12.14
class Derived: public Demo
{
public:
        void inc()
        {
        ++a; // неправильно, поле a у розділі private
        ++b; // правильно, до розділу public мають усі доступ
        ++c; // правильно, нащадок має доступ у розділ protected
        ++d; // неправильно, поле c у розділі private
        }
//правильно, метод inc() має доступ до усіх розділів власного класу
};

int main()
{
        Derived der;
        der.a = 1; // неправильно, поле a у розділі private
        der.b = 2; // правильно, до розділу public мають усі доступ,
                   //у тому числі функція main()
        der.c = 3; //неправильно, функція main() не має доступу до
                   //розділу protected
        der.d = 4; //неправильно, функція main() не має доступу до
                   //розділу private
        der.set();// правильно, метод set() у розділі public
        der.inc();// правильно, метод inc() у розділі public
return 0;
}
```

12.4 Властивості полів

Поля – це змінні, які розташовані у класі. Вони призначені для зберігання даних класу. Говорять, що значення полів визначають стан об’єкта.

Основні властивості:

– поля мають можливість бути любого типу, крім типу власного класу, но можуть бути посиланням чи покажчиком на власний клас. Останнє застосовується для створення списків чи дерев, наприклад:

```
class Tnod { Tnod * next; /* покажчик на власний тип*/};
```

– поля можуть бути *константними*, значення цього поля повинен ініціалізувати конструктор (значення конструктора розглянемо пізніше):

```
class Demo
{
     const int data;
     public: Demo(int v) :data(v) {}
};
```

– поля можуть бути *статичними*. Статичні поля існують в єдиному екземплярі для всіх екземплярів класів і для всієї ієрархії класів.

Статичне поле вимагає обов’язкової ініціалізації:

```
class Demo
{
     static int data;
};
int Demo::data = 20;// ініціалізація
```

Воно доступне як через ім’я об’єкта, так і через тип даних (детальніше розглянемо пізніше).

12.5 Властивості методів

Методи класу – це функції, які розташовані у класі, проте, на відміну від звичайних функцій, мають додатковий ряд властивостей, які будуть детальніше розглянуті в наступних підрозділах.

Метод може бути:

– константним;

– статичним;

– віртуальним/чисто віртуальними;

– спеціальний метод «конструктор»;

– спеціальний метод «деструктор».

Для усіх методів класу працює механізм перевантаження (це коли назва однакова, проте різні параметри), як і для звичайних функцій.

Методи класу можуть перевизначатися у класах-нащадках (це коли і назва, і параметри однакові).

Методи класу (зазвичай public) утворюють інтерфейс, тобто реалізовують механізм для керування об’єктом.

12.6 Покажчик this

Кожен об’єкт має покажчик на себе (на свої дані), який позначається ідентифікатором *this*. Покажчик *this* створюється та ініціалізується автоматично конструктором у момент створення об’єкта.

```
#include<iostream>
class Demo
{
public:
        void printAdrOdj()
        {
std::cout <<this<< std::endl;
//друкуємо адресу розташування об’єкта в пам’яті
        }
};

int main()
{
        Demo d;
        std::cout <<&d << std::endl;
//друкуємо адресу розташування об’єкта в пам’яті
        d.printAdrOdj();
        system("pause");
return 0;
}
```

Результат роботи програми:

```
00EFF9F7
00EFF9F7
```

Зауважимо, що адреси будуть інші, проте однакові, *this* та *&d* – це одна і та ж адреса.

Кожний об’єкт (екземпляр класу) зберігає власний набір полів (крім статичних), проте усі методи класу зберігаються в єдиному екземплярі. Тому, коли викликається метод, йому необхідно знати, з яким із наборів полів йому «працювати», для цього і передається у кожний метод неявно покажчик *this*.

Покажчик *this* можна застосовувати також явно у класі при зверненні до поля чи до методу класу:

```
class Demo
{
        int a, b;
```

```
public:
        void set()
        {
                this->a = 1;
                b = 2;
        }
};
```

У наведеному прикладі перед полем *a* ми вказали покажчик *this*, а перед *b* – ні, хоча компілятор неявно додасть *this->b*.

Покажчик *this* також застосовують явно, коли необхідно повернути «себе» із методу. Цей механізм буде продемонстрований у розділі «перевантаження операцій». Ще одним прикладом застосування *this* є перевірка того, чи є переданий об’єкт тим же об’єктом, який викликав операцію. В наступному прикладі об’єкт *m1* не може нанести собі удар.

```
#include<iostream>
class Monstr
{
public:
        void hit(const Monstr&m)
        {
                if (this == &m)
                        std::cout <<"I can not hit \n";
                else
                        std::cout <<"I hit \n";
        }
};
int main()
{
        Monstr m1, m2;
        m1.hit(m1);
        m1.hit(m2);
        system("pause");
return 0;
}
```

Результат роботи:

```
I can not hit
I hit
```

12.7 Конструктори та деструктори

Після створення об’єкта в пам’яті всі його дані повинні отримати значення, тобто ініціалізуватися. **Конструктор** – це спеціальний метод класу, спеціально призначений для ініціалізації об’єкта.

Основні властивості конструкторів:

- конструктор має те ж ім’я, як і клас;
- не повертає значення (навіть типу void);
- може мати безліч параметрів (за якими поділяється на три типи);
- може бути декілька конструкторів за рахунок перевантаження методів класу;
- не може бути константним, віртуальним, статичним;
- конструктори не успадковуються;
- тільки конструктор може ініціалізувати поля класу (для константних полів ініціалізація обов’язкова);
- якщо програміст не визначив конструктор, транслятор сам будує конструктор за замовчуванням, який не має параметрів і нічого не робить. За наявності в класі інших конструкторів, конструктор без параметрів неявно не створюється.

Клас може мати декілька конструкторів:

- конструкторів за замовчуванням – це конструктори без параметрів;
- конструкторів копіювання (дублювання) – це конструктори з одним параметром: константне посилання на власний тип;
- конструкторів з параметрами – це будь-які інші конструктори, які за параметрами відрізняються від перших двох.
- конструктори переміщення – це конструктори, які приймають *rvalue*-посилання на власний тип і дозволяють "переміщувати" ресурси від одного об'єкта до іншого без зайвого копіювання (додано в стандарті C++11)."

Приклад синтаксису створення конструкторів:

```
class [ім’я_класу]
{
public:
ім’я _класу():
        поле 1 (ініціалізація), поле 2 (ініціалізація), і т.д.
{/* тіло конструктора за замовчуванням*/};
ім’я _класу(const ім’я_класу& ім’я_об’єкта):
        поле 1 (ініціалізація), поле 2 (ініціалізація), і т.д.
{/* тіло конструктора копіювання*/ };
        ім’я _класу(тип ім’я_об’єкта) :
        поле 1 (ініціалізація),поле 2 (ініціалізація), і т.д.
{/* тіло конструктора за параметрами*/ };
ім’я _класу(ім’я_класу&& ім’я_об’єкта):
        поле 1 (ініціалізація), поле 2 (ініціалізація), і т.д.
{/* тіло конструктора переміщення*/ };
};
```

Зауваження:
- ініціалізація обов’язкова тільки для константних полів.
- **&&** – це *rvalue*-посилання, введене зі стандарту C++11. Звичайне посилання & отримало назву *lvalue*. Призначення: *rvalue* посилання дозволяють захоплювати тимчасові об'єкти (*rvalues*) і "переміщувати" їхні ресурси (наприклад, динамічну пам'ять) до інших об'єктів. Це зазвичай робиться для підвищення ефективності, зменшуючи кількість копіювань.

Конструктори викликаються автоматично під час створення об’єкта. Для звичайних об’єктів – коли зустрічається синтаксис:

```
тип ім’я;
```

Для динамічних об’єктів – коли виконується оператор *new*:

```
тип * ім’я = new тип;
```

Деструктор – це особливий вид методу, що застосовується для звільнення пам’яті, зайнятої об’єктом.

Основні властивості деструктора:

- деструктор має те ж саме ім’я, що і клас, перед яким стоїть символ «тильда» (**~**);
- не має аргументів;
- не повертає значення (навіть типу *void*);
- не може бути оголошений як константний чи статичний;
- може бути віртуальним (рекомендовано робити деструктор віртуальним для поліморфних класів).

Якщо деструктор явно не визначений, компілятор автоматично створює порожній деструктор.

Описувати в класі деструктор явно потрібно у разі, коли об’єкт містить покажчики на пам’ять, що виділяється динамічно, інакше при знищенні об’єкта пам’ять, на яку посилалися його поля-покажчики, не буде помічена як вільна.

Деструктор можна викликати явно шляхом вказання повного імені.

Це може знадобитися для об’єктів, яким за допомогою перевантаженої операції *new* виділялася конкретна адреса в пам’яті. Без необхідності не рекомендується явно викликати деструктор об’єкта.

Деструктор викликається автоматично, коли об’єкт виходить з області видимості:

– для локальних об’єктів – при виході з блоку, в якому вони оголошені;

– для глобальних – як частина процедури виходу з головної функції *main()*;

– для динамічних об’єктів деструктор викликається неявно при використанні операції *delete*.

Наприклад, додаємо у клас *Tdate* усі три типи конструкторів та деструктор:

```
#include <iostream>
using namespace std;
class Tdate
{
private:
    int day, month, year;

public:
    Tdate();                    // Конструктор за замовчуванням
    Tdate(const Tdate& obj);    // Конструктор копіювання
    Tdate(int d, int m, int y); // Конструктор з параметрами
    Tdate(Tdate&& obj);         // Конструктор переміщення
    ~Tdate();                   // Деструктор
    void print() const;
};
// Реалізація конструктора за замовчуванням
Tdate::Tdate() : day(1), month(1), year(1900)
{
    cout << "Run default constructor\n";
}
// Реалізація конструктора копіювання
Tdate::Tdate(const Tdate& obj) : day(obj.day), month(obj.month),
year(obj.year)
{
    cout << "Run copy constructor\n";
}
// Реалізація конструктора з параметрами
Tdate::Tdate(int d, int m, int y) : day(d), month(m), year(y)
{
    cout << "Run constructor with parameters\n";
}
// Реалізація конструктора переміщення
Tdate::Tdate(Tdate&& obj) : day(obj.day), month(obj.month),
year(obj.year)
{
    cout << "Run move constructor\n";
}
// Реалізація деструктора
Tdate::~Tdate()
{
    cout << "Run destructor\n";
}
// Метод для виведення дати
void Tdate::print() const
{
    cout << day << "/" << month << "/" << year << endl;
```

```
}
int main()
{
    cout << "Location 1\n";
    {
      Tdate date1;          // Виклик конструктора за замовчуванням
      Tdate date2(31, 12, 2027); // Виклик конструктора з параметрами
        cout << "Location 2\n";
        date1.print();
        date2.print();
        cout << "Location 3\n";
    }
    cout << "Location 4\n";
    {
      Tdate date3(16, 7, 2117); // Виклик конструктора з параметрами
      Tdate date4(date3);       // Виклик конструктора копіювання
      date3.print();     // date3 має значення 0/0/0 після дублювання
      date4.print();
      cout << "Location 5\n";
    }
    {
      cout << "Location 6\n";
      Tdate date5(std::move(Tdate(1, 1, 2000)));
        // Виклик конструктора переміщення
        // Тимчасовий об'єкт створюється та переміщується
        date5.print();
    }
    cout << "Exit\n";
return 0;
}
```

Результат роботи програми:

```
Location 1
Run default constructor
Run constructor with parameters
Location 2
1/1/1900
31/12/2027
Location 3
Run destructor
Run destructor
Location 4
Run constructor with parameters
Run copy constructor
16/7/2117
16/7/2117
Location 5
Run destructor
Run destructor
Location 6
Run constructor with parameters
```

```
Run move constructor
Run destructor
1/1/2000
Run destructor
Exit
```

У наведеному прикладі застосовуються нове поняття «**rvalue-посилання**» – *Tdate&& obj.*

Rvalue (правостороннє значення) в C++ – це тимчасовий об'єкт або літерал, який не має імені і зазвичай не може бути використаний з лівого боку операції присвоювання. Іншими словами, *rvalue* – це значення, яке не прив'язане до певного імені, і його можна використовувати тільки один раз.

Приклад простих *rvalue*:

```
int main() {
    int a = 10;    // 'a' - це lvalue, 10 - rvalue
    int b = a + 5; // 'a + 5' - це rvalue,
                   // результат операції також rvalue
return 0;
}
```

Зазвичай *rvalue* не можна присвоювати, але можна використовувати їх для ініціалізації або передавати як аргументи у функції. У C++ (починаючи з C++11) з'явився новий тип посилань – **rvalue reference** (*T&&*), який дозволяє "захоплювати" тимчасові об'єкти та оптимізувати операції з ними.

```
int main() {

    int&& rvalue_ref = 20;  // 20 - це rvalue
    int&  lvalue_ref = 20;  // невірно, створення lvalue неможливе
return 0;
}
```

Завдяки існуванню *rvalue* ми можемо передавати у функції або методи константи чи тимчасові об’єкти.

У наступному прикладі функція приймає *rvalue*, що дозволяє передати літерал 100.

```
#include <iostream>

void printRvalue(int&& x) {
    std::cout << "Rvalue: " << x << std::endl;
}

int main() {
    printRvalue(100);  // 100 - це rvalue
return 0;
}
```

Цей код викликав би помилку, якщо б замість *rvalue* застосовувалося б *lvalue*.

std::move – це функція, додана в C++11, яка використовується для явного перетворення об'єкта на *rvalue*-посилання. Це дозволяє ініціалізувати інші об'єкти, викликавши конструктор переміщення або оператор переміщення, замість конструктора копіювання.

Основні моменти:

– перетворення на *rvalue*: *std::move* переводить об'єкт у *rvalue*-стан, дозволяючи переміщення ресурсів з одного об'єкта в інший;

– знижує накладні витрати на копіювання великих об'єктів, покращуючи ефективність програми;

– часто застосовується з конструкторами і операторами переміщення для уникнення копіювання.

Оскільки конструктори та деструктори друкують повідомлення, коли вони відпрацювали, є можливість відслідкувати порядок виклику конструкторів та деструкторів для локальних об’єктів.

Порядок виклику конструкторів: спочатку викликаються конструктори полів у порядку їхнього описання в класі, а потім викликається конструктор самого класу.

Порядок виклику деструкторів: деструктори викликаються у порядку, зворотному до виклику конструкторів.

При створені динамічного об’єкта з полями об’єкта працюють через адресу, зазвичай через покажчик, но можливо також створити посилання. Оголошення

```
Tdate*  p;
```

створює покажчик на об’єкт класу *Tdate*. При оголошені покажчика об’єкт не створюється та конструктор в цей момент не відпрацьовує.

Якщо деякий покажчик p вказує на об’єкт, то звернутися до його членів можна за допомогою операцій розадресації (взяття об’єкта за адресою) та операції крапка

```
(*p).метод();
```

або застосувати операцію непрямого вибору **->**, як це застосовувалося для структур

```
p->метод();
```

Покажчику можна присвоїти адресу вже існуючого об’єкта, або створити об’єкт в динамічній пам’яті. Для попереднього прикладу наведемо іншу функцію *main()*, сам клас *Tdate* залишається без змін

```
int main()
{
        cout <<"Location 1\n";
```

```
        Tdate data1(16, 7, 1977);
        cout <<"Location 2\n";
        Tdate * p1 = &data1;
        cout <<"Location 3\n";
        Tdate * p2 = new Tdate(10, 12, 2018); //створення динамічного
                                              //об’єкта
        cout <<"Location 4\n";
        p1->print();
        p2->print();
        cout <<"Location 5\n";
        delete p2; //звільнення динамічного об’єкта
        cout <<"Location 6\n";
        cout <<"Location 7\n";
        cout <<"Exit\n";
        system("pause");
return 0;
}
```

Результат роботи програми:

```
Location 1
Run constructor with parameters
Location 2
Location 3
Run constructor with parameters
Location 4
16/7/1977
10/12/2018
Location 5
Run destructor
Location 6
Run destructor
Location 7
Exit
```

Зауваження:
Двічі видаляти з динамічної пам’яті один і той самий об’єкт заборонено!

Починаючи з C++11, були введені нові ключові слова **default** та **delete**, які дозволили спростити опис стандартної поведінки для конструкторів та деструкторів:

- **default** дозволяє явно вказати, що певна спеціальна функція (наприклад, конструктор, деструктор або оператор присвоєння) повинна мати реалізацію за замовчуванням, яка генерується компілятором. Це корисно, коли ви хочете забезпечити стандартну поведінку для деяких функцій класу, навіть якщо визначаєте інші функції вручну.

- **delete** використовується для заборони виклику певної функції або оператора. Це дає змогу явно зазначити, що певна операція, наприклад, копіювання або переміщення, не повинна бути дозволена для об'єктів даного класу. Це знижує ризик випадкових помилок, коли такі операції використовуються неправильно або взагалі не мають сенсу.

```
#include <iostream>

class Example {
public:
    // конструктор за замовчуванням (згенерований компілятором)
    Example() = default;

    // копіювання об'єкта заборонено
    Example(const Example&) = delete;
    Example& operator=(const Example&) = delete;

    // звичайний конструктор з параметром
    Example(int value) : value_(value) {}

    void print() const {
        std::cout << "Value: " << value_ << std::endl;    }

private:
    int value_ = 0;
};

int main() {
    Example ex1;        // викликається конструктор за замовчуванням
    Example ex2(10);    // викликається конструктор з параметром

    ex1.print();        // output: Value: 0
    ex2.print();        // output: Value: 10

    // спроба копіювання призведе до помилки компіляції
    Example ex3 = ex2;    // помилка: виклик видаленого конструктора
                          // копіювання
    ex1 = ex2;   // помилка: виклик видаленого оператора присвоювання

return 0;
}
```

Раніше, щоб заборонити виклик певного конструктора або оператора присвоювання, ці функції переміщували в розділ *private*, наприклад:

```
class Example {
private:
    MyClass(const MyClass&);            // забороняємо копіювання
    MyClass& operator=(const MyClass&); // забороняємо присвоювання
                                        // копіюванням
};
```

У цьому випадку, хоча ці функції недоступні поза межами класу, компілятор все одно генерує їхню реалізацію.

Як відомо, в C++ існує неявне приведення типів, що іноді може призводити до небажаних наслідків або помилок. Щоб уникнути таких ситуацій, було введено ключове слово **explicit**, яке дозволяє заборонити неявне приведення типів під час виклику конструктора або оператора конверсії.

```
#include <iostream>
using namespace std;
class NoExpliticClass {
public:
    NoExpliticClass() : value(0) {    }
    NoExpliticClass(int x) { value = x;
    }
    void print() const
{std::cout << "Value: " << value << std::endl;    }

private:
    int value;
};

int main() {
    NoExpliticClass obj1(10);  // коректно: явне приведення
    obj1.print();      // output: Value: 10
    NoExpliticClass obj2 = 20;  // коректно: неявне приведення
    obj2.print();      // output: Value: 20
    NoExpliticClass obj3;
    obj3 = 30;          // коректно: неявне приведення
    obj3.print();       // output: Value: 30
return 0;
}
```

При оголошенні конструктора як неявне приведення буде заборонено і викликати помилку:

```
#include <iostream>
using namespace std;
class ExpliticClass {
public:
    ExpliticClass() : value(0) {
    }
    explicit ExpliticClass(int x) {
        value = x;
    }
    void print() const {
        std::cout << "Value: " << value << std::endl;
    }
private:
    int value;
```

```
};
int main() {
    ExpliticClass obj1(10);  // неправильно: явне приведення
    obj1.print();      // output: Value: 10
    ExpliticClass obj2 =20;  // неправильно: неявне приведення
    ExpliticClass obj3;
    obj3 = 30;          // неправильно: неявне приведення
return 0;
}
```

12.8 Статичні елементи класу

За допомогою модифікатора *static* можна описати статичні поля та методи класу. Їх можна розглядати як глобальні змінні або функції, що доступні тільки в межах області класу.

Статичні поля застосовуються для зберігання даних, загальних для всіх об’єктів класу, наприклад, кількості об’єктів або посилання на ресурс, спільний для всіх об’єктів.

Ці поля існують для всіх об’єктів класу в єдиному екземплярі, тобто не дублюються.

Особливості статичних полів

Пам’ять під статичне поле виділяється один раз при його ініціалізації незалежно від числа створених об’єктів.

Статичні поля доступні як через ім’я класу, так і через ім’я об’єкта.

На статичні поля поширюється дія специфікаторів доступу.

Пам’ять, яку займає статичне поле, не враховується при визначенні розміру об’єкта за допомогою операції *sizeof()*.

```
#include<iostream>
using namespace std;
class A {
public:
        static int cout; //оголошення статичного поля
};
int A::cout=10; //ініціалізація статичного поля
A b, *c;
int main()
{
        cout <<A::cout << endl;          //10
        cout << b.cout << endl;          //10
        cout << c->cout << endl;         //10
        A::cout = 100;
        cout <<A::cout << endl;          //100
        cout << b.cout << endl;          //100
        cout << c->cout << endl;         //100
```

```
return 0;
}
```

Результат роботи:

```
10
10
10
100
100
100
```

12.9 Константні методи

Константні методи – це методи, які не мають права змінювати дані свого класу. Говорять, що константні методи не змінюють стан об’єкта. Для створення константного методу необхідно після опису параметрів вказати ключове слово *const*. Константні методи мають право викликати тільки константні методи та не мають права викликати неконстантні. Наприклад:

```
class D
{
        int data;
public:
        int getData() const {return data; } //константний метод
};
```

12.10 Агрегація та композиція

В ООП існує два поняття «агрегація» та «композиція» – коли полями класу є об’єкти інших класів. Залежно від того, чи існували поля до створення об’єкта, говорять по агрегацію чи композицію.

Композиція (англ. *composition*) – це коли поля не існували до створення об’єкта і вони існують, доки існує сам об’єкт.

Агрегація (англ. *aggregation*) – це коли поля існували і до створення об’єкта, вони можуть продовжити існування і після руйнування об’єкта.

Наприклад, квартира складається з кімнат, у цьому випадку маємо композицію, тому що кімната без квартири не існує.

Інший приклад – одна й та ж сама людина може бути членом різних спільнот за інтересами, наприклад, «Любителі кішок». Проте, навіть якщо ця спільнота розпадеться, її учасники продовжать існувати та залишаться учасниками інших спільнот. У мові C++ композиція реалізовується у вигляди полів-покажчиків чи полів-

посилань, які отримують адреси зовнішніх об’єктів. У інших випадках ми говоримо, що виконується агрегація.

Наведемо приклад. Створимо лінію, яка складається з двох об’єктів «крапка» (*Point*). Клас *LineComposition* демонструє композицію, а *LineAggregation* – агрегацію.

```
#include<iostream>
class Point
{
    int x, y;
public:
    Point(int x_, int y_) : x(x_), y(y_)
    {
        std::cout << "Create point " << x << "," << y << "\n";
    }
    ~Point()
    {
        std::cout << "Delete point " << x << "," << y << "\n";
    }
    void print() const
    {
        std::cout << "Point " << x << "," << y << "\n";
    }
};
class LineComposition //клас, який реалізований через композицію
{
    Point p1, p2;
public:
    LineComposition(int x1, int y1, int x2, int y2) :p1(x1, y1),
p2(x2, y2)
    {
        std::cout << "Create line \n";
        p1.print();
        p2.print();
    }
    ~LineComposition()
    {
        std::cout << "Delete line \n";
        p1.print();
        p2.print();
    }
    void print() const { p1.print();    p2.print(); }
};

class LineAggregation//клас, який реалізований через агрегацію
{
    const Point& p1;
    const Point& p2;
public:
```

```
    LineAggregation(const Point& p1_, const Point& p2_) :p1(p1_),
p2(p2_)
    {
        std::cout << "Create line \n";
        p1.print();
        p2.print();
    }
    ~LineAggregation()
    {
        std::cout << "Delete line \n";
        p1.print();
        p2.print();
    }
    void print() const { p1.print();     p2.print(); }
};

int main()
{
    {
        std::cout << "Location 1\n";
        LineComposition lineC(10, 20, 30, 40);
        lineC.print();
    }
    std::cout << "\nLocation 2\n";
    Point point1(11, 21), point2(31, 41);
    {
        std::cout << "\nLocation 3\n";
        LineAggregation lineA(point1, point2);
        lineA.print();
    }
    std::cout << "\nLocation 4\n";
    point1.print();
    point2.print();
    std::cout << "Exit\n";
return 0;
}
```

Результат роботи:

```
Location 1
Create point 10,20
Create point 30,40
Create line
Point 10,20
Point 30,40
Point 10,20
Point 30,40
Delete line
Point 10,20
Point 30,40
Delete point 30,40
Delete point 10,20
```

```
Location 2
Create point 11,21
Create point 31,41

Location 3
Create line
Point 11,21
Point 31,41
Point 11,21
Point 31,41
Delete line
Point 11,21
Point 31,41

Location 4
Point 11,21
Point 31,41
Exit
Delete point 31,41
Delete point 11,21
```

Як видно з результатів роботи, поля об’єкта класу *LineComposition* існують, поки існує сам об’єкт, проте об’єкти *point1* та *point2* існують і після того, як об’єкт *LineAggregation* був вилучений з пам’яті.

12.11 Статичні методи

Статичні методи призначені для звернення до статичних полів класу, або оформлення загального коду, який не залежить від нестатичних полів (не застосовує їх). Такі методи можуть звертатися безпосередньо тільки до статичних полів або інших статичних методів, оскільки їм не передається прихований покажчик *this*. Звернення до статичних методів проводиться так само, як до статичних полів – або через ім’я класу, або, якщо хоча б один об’єкт класу вже створений, через ім’я об’єкта.

```
#include<iostream>
using namespace std;
class B {
    static int count;
    int data;

public:
    static void inc_count() { count++; }
    static int get_count() { return count; }
};
int B::count(13);
int main()
```

```
{
    B d, * e;
    cout << d.get_count() << endl;
    d.inc_count();
    cout << e->get_count() << endl;
return 0;
}
```

Результати роботи:

```
13
14
```

Статичні методи не можуть бути константними (*const*) і віртуальними (*virtual*).

Наступний приклад демонструє, як можна пронумерувати усі створювані об’єкти (надати унікальний номер кожному об’єкту). Поле *total_count* відповідатиме за загальну кількість створених об’єктів (незалежно від уже видалених), *current_count* – кількість об’єктів, які існують в поточний час, і поле *number* – це унікальний номер об’єкта.

```
#include <iostream>
using namespace std;
class Tsequence
{
private:
static int total_count; //загальна кількість створених об’єктів
static int current_count;//кількість об’єктів,
                         //що існують в поточний час
        const int number;//унікальний номер об’єкта
public:
        Tsequence();     //конструктор за замовчуванням
        Tsequence(const Tsequence& obj);//конструктор копіювання
        Tsequence(int d, int m, int y); //конструктор з параметрами
        ~Tsequence();                   //деструктор
        static int getTotalCount() { return total_count; };
        static int getCurrentCount() { return current_count; };
        int getNumber()                 const { return number; };

};
Tsequence::Tsequence() : number(total_count)
{
cout << "Run default constructor: Create object with number = ";
                cout << number << "\n";
        ++total_count;
        ++current_count;
}
Tsequence::Tsequence(const Tsequence& obj) : number(total_count)
{
cout << "Run copy constructor: Create object with number = ";
        cout << number << "\n";
```

```
        ++total_count;
        ++current_count;
}
Tsequence::~Tsequence()
{
cout<<"Run destructor: Deleting object with number = "<<number<<"\n";
        --current_count;
}

int Tsequence::total_count = 0;
int Tsequence::current_count = 0;

void functionWithByValue(Tsequence obj)
{
        cout << "In function functionWithByValue()\n";
        cout << " Object with number = " << obj.getNumber() << endl;
}
int main()
{
        cout << "Location 1\n";
        Tsequence seq1;
        {
        Tsequence seq2;
        Tsequence seq3;
        Tsequence seq4;
        cout << "Location 2\n";
cout <<" Total objects = "<< Tsequence::getTotalCount() << endl;
cout <<" Current exists objects = ";
cout << Tsequence::getCurrentCount()<< endl;
        functionWithByValue(seq4);
        cout << "Location 3\n";
cout << " Total objects = "<< Tsequence::getTotalCount() << endl;
cout << " Current exists objects = ";
cout << Tsequence::getCurrentCount() << endl;
        }
        cout << "Location 4\n";
cout << " Total objects = "<< Tsequence::getTotalCount() << endl;
cout << " Current exists objects = ";
cout << Tsequence::getCurrentCount() << endl;

        cout << "Exit\n";
return 0;
}
```

Результат роботи програми:

```
Location 1
Run default constructor: Create object with number = 0
Run default constructor: Create object with number = 1
Run default constructor: Create object with number = 2
Run default constructor: Create object with number = 3
Location 2
```

```
 Total objects = 4
 Current exists objects = 4
Run copy constructor: Create object with number = 4
In function functionWithByValue()
 Object with number = 4
Run destructor: Deleting object with number = 4
Location 3
 Total objects = 5
 Current exists objects = 4
Run destructor: Deleting object with number = 3
Run destructor: Deleting object with number = 2
Run destructor: Deleting object with number = 1
Location 4
 Total objects = 5
 Current exists objects = 1
Exit
Run destructor: Deleting object with number = 0
```

12.12 Дружні функції і класи

Якщо необхідно дозволити деякій функції чи методу іншого класу прямий доступ до розділів *private* чи *protected*, таку функцію чи метод необхідно оголосити як **дружню**. Не треба зловживати дружніми функціями. Бажано доступ до полів все ж таки реалізовувати через методи класу.

Дружні функції застосовуються для доступу до прихованих полів класу і є альтернативою методам, проте дружні функції не є інтерфейсом класу.

Правила опису та особливості дружніх функцій:

– дружня функція оголошується всередині класу, до елементів якого їй потрібен доступ, з ключовим словом **friend**;

– одним із параметрів до дружньої функції повинен передаватися об’єкт класу або адреса об’єкта класу, оскільки покажчик *this* дружній функції не передається;

– дружня функція може бути звичайною функцією або методом іншого раніше визначеного класу.

Одна функція може бути дружньою відразу декільком класам.

Оголошення *friend* не є специфікатором доступу і не успадковується.

Наведемо приклад дружньої функції:

```
class A
{
private:
        int data;
        friend void f(A&a);
};
```

```
void f(A&a)
{ a.data = 20; }
```

Дружній клас

Якщо усім методам деякого класу *A* необхідно надати права доступу до усіх розділів іншого класу *B*, такий клас необхідно оголосити як дружній. Для цього необхідно в будь-якій частини слова класу *A* розташувати наступний елемент *friend class B;*

Тобто:

```
class B;
class A
{
        friend class B;
};
```

Наступний приклад демонструє застосування дружнього класу та методу:

```
#include<iostream>
using namespace std;
class TA {
        int a;
        int b;
public:
        int d;
        TA(int x = 3, int y = 4) { a = x; b = y; }
        friend void print(TA&);
        friend class TB;
};
class TB {
public:
        void inc_a(TA & inca) { inca.a++; };
        void inc_b(TA& incb) { incb.b++; };
};
void print(TA& obj)
{
        cout << obj.a << "\t" << obj.b << endl;
}

int main()
{
        TA b;   print(b);
        TB q;
        q.inc_a(b);
        q.inc_b(b);
        print(b);
return 0;
}
```

Результат роботи:

```
3       4
4       5
```

12.13 Перевизначення операцій

C++ дозволяє перевантажити чи визначити дію більшості операцій так, щоб при використанні з об’єктами конкретного класу вони виконували власні дії. Це дає можливість зручно використовувати власні типи даних таким же чином, як і стандартні. Проте не всі операції є можливість перевизначити. Наступні операції заборонено перевантажувати:

?: :: # ## sizeof

Перевантаження операцій здійснюється за допомогою методів спеціального виду «метод-операція» або зовнішньою функцією і підпорядковується таким правилам:

– при перевантаженні операцій зберігається кількість аргументів, пріоритети операцій і правила асоціації (справа наліво або зліва направо), які використовуються в стандартних типах даних;

– операції не можуть мати аргументів за замовчуванням;

– операції успадковуються (за винятком =);

– операції не можуть визначатися як *static*;

– для стандартних типів даних перевизначити операції неможливо.

Загальний синтаксис перевантаження операції має такий вигляд (за винятком перевантаження приведення до типу):

```
тип operator операція (список параметрів) { тіло функції }
```

Перевантаження операції приведення до типу має інший вигляд:

```
operator операція (список параметрів) { тіло функції }
```

Перевантаження більшості операцій може бути виконано як методом класу/структури, так і зовнішньою функцією. При перевантаженні операції методом класу/структури, першим аргументом є сам клас/структура. У цьому випадку необхідно передавати тільки другий аргумент (якщо він є). При перевантаженні операції зовнішньою функцією всі аргументи необхідно передавати як параметри.

Зауваження:
Операція приведення до типу перевантажується тільки всередині класу.

Зауваження:
Якщо необхідно перевантажити операцію для різнотипних аргументів і перший аргумент повинен відрізнятися від типу класу, таку операцію перевантажують тільки зовнішньою функцією. Класичним прикладом такого перевантаження є перевантаження потокових операцій.

Проілюструємо перевантаження деяких операторів для наступних типів, створених користувачем:

```
struct Tstud
{
        char pib[20];
        int bal;
};
struct TAB
{
        float A;
        int B;
};
```

Перевантаження унарних операцій зовнішньою функцією

Перевантажимо префіксну операцію "**++**" зовнішньою функцією, а операцію "−" методом структури.

Оскільки операція унарна, як аргументи передається один параметр.

Тепер для змінних (об’єктів) типу *Tstud* можна використовувати операції інкременту та декременту:

```
#include<iostream>
using namespace std;
struct Tstud
{
    char pib[20];
    int bal;
    Tstud& operator --()// перевантаження методом класу
    {
        --bal;
        return (*this);
    }
};
Tstud& operator ++(Tstud& argument) //перевантаження зовнішньою
                                    // функцією
{
    ++argument.bal;
    return argument;
}

int main()
{
    Tstud student = { "Petrenko",3 };
```

```
    ++student;  // Student={"Petrenko",4}
    cout << student.bal << endl;
    --student;  // Student={"Petrenko",2}
    cout << student.bal << endl;
return 0;
}
```

Результат роботи програми:

```
4
3
```

Для **перевантаження постфіксних** унарних операцій необхідно як параметр передати також фіктивний параметр *int*. За рахунок нього компілятор розрізнятиме префіксну та постфіксну форми.

При перевантаженні постфіксної операції рекомендується дотримуватися такого алгоритму:

– створити локальний об’єкт, як копію аргументу (всередині класу/структури бажано через конструктор копіювання);

– викликати префіксну операцію;

– повернути (через *return*) збережений об’єкт на першому кроці.

Продемонструємо це на прикладі. Розширимо попередній приклад постфіксними операціями. Аналогічно продемонструємо перевантаження методом класу і зовнішньою функцією.

```
#include<iostream>
#include<string.h>
using namespace std;

struct Tstud
{
        string pib;
        int bal;
// перевантаження методом класу префіксної форми
        const Tstud& operator --()
        {
                --bal;
                return (*this);
        }
// перевантаження методом класу постфіксної форми
        Tstud operator--(int)
        {
                Tstud res;
                res.pib = this->pib;
                res.bal = this->bal;
                --(*this);
                return (res);
        }

};
```

```
//перевантаження зовнішньою функцією
const Tstud& operator ++(Tstud& argument)
{
        ++argument.bal;
        return argument;
}
// перевантаження постфіксної форми зовнішньою функцією
Tstud operator++(Tstud & argument, int)
{
        Tstud res;
        res.pib = argument.pib;
        res.bal = argument.bal;
        ++argument;
        return (res);
}
int main()
{
        Tstud student1 = { "Petrenko",3 };
        Tstud student2 = { "Petrenko",3 };
        Tstud student3;
        Tstud student4;
        student3 = ++student1;
        student4 = student2++;
        cout << student1.bal << endl;
        cout << student2.bal << endl;
        cout << student3.bal << endl;
        cout << student4.bal << endl;
return 0;
}
```

Результат:

```
4
4
4
3
```

Як можна побачити, змінна *student4* отримала поточне значення змінної *student3* (три), а тільки потім змінна *student3* змінила своє значення (на чотири).

Перевантаження бінарних операцій

Більшість бінарних операцій мають споріднену операцію з символом `=`, наприклад, операція `+` і `+=`. При перевантаженні таких операцій рекомендується спочатку перевантажувати операцію `+=`, а потім застосовувати її при перевантаженні операції `+`.

Перевантажимо ці операції для структури *Tab*:

```
#include<iostream>
using namespace std;
```

```
struct Tab
{
        float a;
        int b;
        Tab() :a(0.0), b(0) {}
        Tab(double _a, int _b) :a(_a), b(_b) {}
        Tab(const Tab& obj) :a(obj.a), b(obj.b) {}
//перевантаження методом класу
        Tab& operator +=(const Tab& argument2)
        {
                this->a += argument2.a;
                this->b += argument2.b;
                return (*this);
        }
//перевантаження методом класу
        Tab operator + (const Tab & argument2) const
        {
                Tab res(*this);
                res += argument2; //викликаємо попередню операцію +=
                return (*this);
        }
        // перевантаження різнотипної операції Tab+double
        Tab& operator +=(double argument2) //перевантаження методом
                                           // класу
        {
                this->a += argument2;
                this->b += argument2;
                return (*this);
        }
        Tab operator + (double argument2) const //перевантаження
                                                //методом класу
        {
                Tab res(*this);
                res += argument2;//Викликаємо попередню операцію +=
                return (*this);
        }

};

//перевантаження зовнішньою функцією
Tab& operator -=(Tab& argument1, const Tab& argument2)
{
        argument1.a += argument2.a;
        argument1.b += argument2.b;
        return argument1;
}
//перевантаження зовнішньою функцією
Tab operator - (const Tab & argument1, const Tab & argument2)
{
        Tab res(argument1);
        res += argument2;//Викликаємо попередню операцію +=
```

```
        return (res);
}

// перевантаження різнотипної операції double+ Tab
//таку операцію можна перевантажити тільки ззовні,
//тому що перший аргумент відрізняється від типу класу (Tab),
//для якого перевантажується операція

double& operator +=(double& argument1, const Tab& argument2)
//перевантаження методом класу
{
        argument1 += (argument2.a + argument2.b);
        return (argument1);
}
double operator + (double argument1, const Tab & argument2)
//перевантаження методом класу
{
        double res(argument1);
        res += argument2;       //викликаємо попередню операцію +=
        return (res);
}

int main()
{
        Tab ab1(1.5, 1), ab2(2.7, 2), ab3;
        cout << "ab1 = " << ab1.a << "\t" << ab1.b << endl;
        ab1 += ab2;
        cout << "ab1 = " << ab1.a << "\t" << ab1.b << endl;
        ab1 += 5.0;
        cout << "ab1 = " << ab1.a << "\t" << ab1.b << endl;
        ab3 = ab1 + ab2;
        cout << "ab1 = " << ab1.a << "\t" << ab1.b << endl;
        double res = 0;
        res += ab3;
        cout << "res = " << res << endl;
        ab2 += 10;
        cout << "ab2 = " << ab2.a << "\t" << ab2.b << endl;
        ab3 = ab1 - ab2;
        cout << "ab3 = " << ab3.a << "\t" << ab3.b << endl;
return 0;
}
```

Результат:

```
ab1 = 1.5       1
ab1 = 4.2       3
ab1 = 9.2       8
ab1 = 9.2       8
res = 17.2
ab2 = 12.7      12
ab3 = 21.9      20
```

Тепер перевантажимо бінарні операції "**>**", "**<**" для структури *Tab*. Вважатимемо, що лівий операнд «більше» правого, якщо суми їхніх полів більші. Результат цієї операції має бути логічного типу:

```
#include<iostream>
using namespace std;


struct Tab
{
        float a;
        int b;
        Tab() :a(0.0), b(0) {}
        Tab(double _a, int _b) :a(_a), b(_b) {}
        Tab(const Tab& obj) :a(obj.a), b(obj.b) {}
        bool operator > (const Tab& argument2)
        {
        return (this->a + this->b) > (argument2.a + argument2.b);
        }

};
bool operator < (const Tab& argument1, const Tab& argument2)
{
return (argument1.a + argument1.b) < (argument2.a + argument2.b);
}

int main()
{
        Tab ab1(1.5, 1), ab2(2.7, 2);
        cout << "ab1 > ab2 =" << (ab1 > ab2) << endl;
        cout << "ab1 < ab2 =" << (ab1 < ab2) << endl;
return 0;
}
```

Результат:

```
ab1 > ab2 =0
ab1 < ab2 =1
```

Перевантаження операції приведення до типу

Можна визначити функції-операції, які здійснюватимуть приведення об’єкта класу до іншого типу. Формат:

```
operator ім’я_нового_типу ();
```

Тип значення, що повертається і параметри вказувати не потрібно. Операцію приведення до типу можна визначити тільки всередині класу/структури.

Перевантаження операції функціональних дужок має такий синтаксис:

```
тип operator () (список параметрів);
```

Перевантаження функціональних дужок для одного класу може бути безліч для різних параметрів, за типом чи кількістю (працює механізм перевантаження методів).

Клас чи структура, у якій визначені функціональні дужки, називають **функтором** чи функціональним об’єктом, тому що такий об’єкт можна застосовувати як функцію.

Для прикладу визначимо оператор приведення до типу int і визначимо оператор функціональних дужок, який задає поля структури:

```
#include<iostream>
#include<string>
using namespace std;
struct Tstud
{       string pib;
        int bal;
        operator int() const { return (bal); }
        operator const char* () const { return pib.c_str(); }
        void operator()(const char* str, int mybal)
        {       pib =  str;
                bal = mybal;
        }
        void operator()() const
        {
                cout << pib << "\t" << bal << endl;
        }
};

int main()
{
        Tstud st;
        st("Petrenko", 75);
        st();
        cout << "Bal = " << (int)st << endl;
        cout << "pib = " << (const char*)st << endl;
return 0;
}
```

Результат роботи:

```
Petrenko  75
Bal = 75
pib = Petrenko
```

Потоки і типи, визначені користувачем

Для введення і виведення в потоках використовуються перевантажені для всіх стандартних типів операції виводу у потік і зчитування з потоку, відповідно «**<<**» і «**>>**». При цьому вибір конкретної операції визначається типом фактичних параметрів. Для того, щоб вводити і виводити величини типів, визначених користувачем, необхідно перевантажити ці операції. Це бінарні операції, лівим операндом яких є об’єкт-потік, а правим – об’єкт, який потрібно вилучити або помістити в цей потік. Значення, що повертається, має бути посиланням на потік, щоб можна було створювати ланцюжок операцій, як і у випадку зі стандартними типами.

Синтаксис визначення власної функції виводу (поміщення) в потік:

```
ostream& operator << (ostream& ім’я_потоку, constТип& об’єкт)
{
        тіло функції вставки
                return ім’я_потоку;
}
```

Формат визначення власної функції читання (видалення) з потоку:

```
istream& operator >> (istream& ім’я_потоку, Тип& об’єкт)
{
        тіло функції вставки
                return ім’я_потоку;
}
```

Визначимо власні функції виводу і введення у потоки для структури *Tstud*:

```
#include<iostream>
#include<iomanip>
#include<string>

using namespace std;

struct Tstud
{
        string name;
        int bal;
        operator int() const { return (bal); }
        operator const char* () const { return (name.c_str()); }
};

ostream& operator<<(ostream& out, const Tstud& stud)
{
out << setw(10) << setfill('.') << setiosflags(ios::left);
out << stud.name << stud.bal << endl;
        return out;
```

```
}

istream& operator>>(istream& in, Tstud& stud)
{
        cout << "Input Name\n";
        in >> stud.name;
        cout << "input Bal\n";
        in >> stud.bal;
        return in;
}


int main()
{
        Tstud st;
        cin >> st;
        cout << st;
return 0;
}
```

Приклад роботи програми:

```
Input Name
Petrenko
input Bal
96
Petrenko..96
```

Перевантаження операції «=»

Операція «=» – це єдина операція, яка автоматично визначена для структури/класу. Визначена вона за замовчуванням як побітове копіювання об’єкта. Це може призвести до фатальних наслідків, якщо в класі застосовується динамічна пам’ять, а саме: в конструкторі вона виділяється, а в деструкторі звільняється. У випадку копіювання об’єктів, створених від такого типу, операція «=» за замовчуванням скопіює адрес пам’яті, а не елементи з динамічної пам’яті. Тому для таких класів операцію «=» необхідно перевизначити.

Рекомендується такий алгоритм:

– перевіряємо, що об’єкт «не дублює самого в себе» a = a (this == &obj). Якщо так, тоді достроковий вихід;

– звільняємо динамічну пам’ять (рекомендовано викликати деструктор);

– дублюємо нединамічні поля класу;

– виділяємо нову динамічну пам’ять необхідного розміру;

– копіюємо елементи з динамічної пам’яті.

Зауваження:
Цей алгоритм звичайно може бути оптимізований, наприклад, коли розмір обох об’єктів однаковий, то немає потреби перевиділяти динамічну пам’ять.

Розглянемо це на прикладі (більш повна версія з результатом буде наведена пізніше):

```
class Tvec
{
        int *vec;
        int n;

public:
        Tvec(int n);// конструктор з параметром
        ~Tvec() { delete[] vec; vec = NULL; }//деструктор
        // перевантаження операції "="
        Tvec&operator=(const Tvec&);
};
Tvec& Tvec::operator=(const Tvec& obj)
{
        if (this == &obj) return *this;
        this->~Tvec();
        n = obj.n;
        vec = new int[n];
        for (int i(0); i<n; i++)
                vec[i] = obj.vec[i];
        return *this;
}
```

Перевантаження операції індексації «[]» дає можливість звертатися до деяких елементів класу як до елементів звичайного масиву: *mas[i]*.

Перевантаження операції індексації необхідно виконувати в двох виглядах:

- не константній формі для підтримки операції запису: *mas[i]=value*;
- константній формі для підтримки операції читання: *value = mas[i]*.

Синтаксис обох форм перевантаження операції у класі відповідно:

```
// перевантаження операції індексації "[]"
Тип_елемента_масиву&operator[](int index);
// перевантаження операції індексації "[]"
Тип_елемента_масиву&operator[](int index)const;
```

Як видно з синтаксису, операція вимагає наявності одного параметру, який відповідає за індекс доступу до елемента. У програміста є можливість перевірити коректність переданого індексу.

У випадку, якщо індекс не попадає в інтервал масиву, зазвичай реалізовують одну з двох поведінок:

– генерують виключення (**exception**);

– повертають не посилання елемента з масиву, а посилання на додатково створений об’єкт. У цьому випадку говорять про заглушення неправильного індексу.

Перевантаження індексації (а також більшість попередньо розглянутих операції) продемонстровано у наступному прикладі «Вектор».

Приклад 12.1.

Клас "Вектор" реалізовує конструктори за замовчуванням, з параметром і конструктор копіювання. Реалізовано у вигляді методів заповнення масиву випадковим чином; знаходження суми елементів масиву.

Перевантажені такі операції:

– порівняння (повертає істину, якщо сума елементів обох операндів рівна);

– присвоювання;

– потокові операції введення «>>» і виведення «<<».

Проілюстровано роботу реалізованих методів.

```
#include<iostream>
#include<time.h>
class Tvec
{
        int* vec;
        int n;
public:
        Tvec();//конструктор за замовчуванням
        Tvec(int n);// конструктор з параметром
        Tvec(const Tvec&);//конструктор копіювання
        ~Tvec() { delete[] vec; vec = NULL; }//деструктор
// метод заповнення випадковим чином від 0 до заданого параметра
        void randmas(int);
        int sum()const;//метод знаходження суми елементів
        bool operator == (const Tvec&)const;// перевантаження
                                            //операції "=="
        Tvec& operator=(const Tvec&);// перевантаження операції "="
        Tvec operator + (const Tvec&)const;// перевантаження операції "+"
        // перевантаження потоків
        // перевантаження операції індексації"[]"
        const int& operator[](int index)const;
        int& operator[](int index);// перевантаження операції
                                   // індексації "[]"
        int size()const { return n; }
private:
        int value_by_bad_index;
```

```
};

std::istream& operator>>(std::istream& in, Tvec& obj);
std::ostream& operator<<(std::ostream& out, const Tvec& obj);

int main()
{
        Tvec vec1, vec2(10);
        vec2.randmas(5);
        vec1.randmas(10);
        std::cout << "Pochatkovi dani:" << std::endl;
        std::cout << "vec1 = " << vec1 << std::endl;
        std::cout << "vec2 = " << vec2 << std::endl;
        std::cout << "Sum1 = " << vec1.sum() << std::endl;
        std::cout << "Sum2 = " << vec2.sum() << std::endl;
if (vec1 == vec2)   std::cout << "vec1==vec2" << std::endl;
else    std::cout << "vec1!=vec2" << std::endl;
        Tvec vec3(vec2);
if (vec3 == vec2)   std::cout << "vec3==vec2" << std::endl;
else    std::cout << "vec3!=vec2" << std::endl;
        vec1 = vec2;
        std::cout << "Vec1 = " << vec1 << std::endl;
        vec3.randmas(8);
        std::cout << "Vec3 = " << vec3 << std::endl;
        std::cout << "Vec1+Vec3 = " << vec1 + vec3 << std::endl;
        system("pause");
return 0;
}

Tvec::Tvec()
{
        this->n = 5;
        vec = new int[n];
}

Tvec::Tvec(int n)
{
        this->n = n;
        vec = new int[n];
}

Tvec::Tvec(const Tvec& obj)
{
        this->n = obj.n; vec = new int[n];
        for (int i(0); i < n; i++)
                vec[i] = obj.vec[i];
};
void Tvec::randmas(int k)
{
        srand(time(NULL));
        for (int i(0); i < n; i++)
```

```
                vec[i] = rand() % k;
}
int Tvec::sum()const
{
        int s(0);
        for (int i(0); i < n; i++)
                s += vec[i];
        return s;
}
bool Tvec::operator==(const Tvec& obj) const
{
        int s1 = this->sum();
        int s2 = obj.sum();
        return (s1 == s2);
}
Tvec& Tvec::operator=(const Tvec& obj)
{
        if (this == &obj) return *this;
        this->~Tvec();
        n = obj.n;
        vec = new int[n];
        for (int i(0); i < n; i++)
                vec[i] = obj.vec[i];
        return *this;
}
Tvec Tvec::operator+(const Tvec& obj) const
{
        int m = n + obj.n;
        Tvec Merg(m);
        int i(0);
        for (; i < n; i++)
                Merg.vec[i] = vec[i];
        for (; i < m; i++)
                Merg.vec[i] = obj.vec[i - n];
        return Merg;
}
std::istream& operator>>(std::istream& in, Tvec& obj)
{
        for (int i(0); i < obj.size(); i++)
                in >> obj[i];
        return in;
}
std::ostream& operator<<(std::ostream& out, const Tvec& obj)
{
        for (int i(0); i < obj.size(); i++)
                out << obj[i] << " ";
        out << '\n';
        return out;
}
const int& Tvec::operator[](int index)const
{
```

```
//Якщо індекс неправильний - повертаємо посилання на змінну
//value_by_bad_index
        return (index < this->n) ? vec[index] : value_by_bad_index;
}
int& Tvec::operator[](int index)
{
//Якщо індекс неправильний - повертаємо посилання на змінну
//value_by_bad_index
        return (index < this->n) ? vec[index] : value_by_bad_index;
}
```

Приклад виконання програми:

```
Pochatkovi dani:
vec1 = 9 4 0 4 4

vec2 = 4 4 0 4 4 2 4 4 2 3

Sum1 = 21
Sum2 = 31
vec1!=vec2
vec3==vec2
Vec1 = 4 4 0 4 4 2 4 4 2 3

Vec3 = 1 2 2 0 2 6 3 1 7 5

Vec1+Vec3 = 4 4 0 4 4 2 4 4 2 3 1 2 2 0 2 6 3 1 7 5
```

12.14 Наслідування

Похідний клас (нащадок) може отримувати атрибути та поведінку вже існуючого батьківського (або базового) класу. **Наслідування** дозволяє винести щось спільне в батьківський клас, що притаманне деяким класам. Загальні властивості задаються в базовому класу, а відмінності – його наступникам. Клас нащадок може бути створений від декількох базових класів, так зване *множинне наслідування*.

Загальний синтаксис створення класу нащадку такий:

```
class[ім’я_кл_нащадку]:[public|protected|private][базовий клас1],
                       [public|protected|private][базовий клас2], …
                       [public|protected|private][базовий класN]
{   [public:]     // доступно всім
[дані, методи, властивості, події]
[protected:]      // доступний тільки нащадкам
[дані, методи, властивості, події]
[private:]        // доступні тільки у класі
[дані, методи, властивості, події]
}
 [список об’єктів];
```

Наприклад, створимо базовий клас *Point* і похідний *ColorPoint*. У базовому класі зберігаємо інформацію про координати точки, а у нащадку додаємо інформацію про колір:

```
#include<iostream>
class Point
{
protected:
        int x_, y_;
public:
        Point(int x, int y) :x_(x), y_(y) {}
        void print() const
        {std::cout << x_ << " " << y_<<"\n";    }
};
class ColorPoint: public Point
{
        int color_;
public:
        ColorPoint(int x, int y, int c) :Point(x, y), color_(c) {}
        void print() const
        {
                Point::print();
                std::cout << x_ << " " << y_ << " " << color_;
        }
};
int main()
{
        ColorPoint cp(10, 20, 255);
        cp.print();
return 0;
}
```

Результат роботи:

```
10 20
10 20 255
```

Конструктори не наслідуються, мають на увазі, що в класі нащадку необхідно створювати власні конструктори (примітка: зазвичай усі конструктори базового класу будуть у наявності у класі нащадку, проте конструктори самого класу нащадку автоматично не з’являться).

При створенні об’єкта від класу нащадку обов’язково викликатиметься спочатку конструктор базового класу, а потім конструктор класу нащадку. Якщо базових класів декілька, то викликатимуться конструктори всіх базових класів у порядку описання а потім конструктор класу нащадку. Деструктори викликаються у зворотному порядку, тобто спочатку деструктор класу нащадку, а потім деструктори базових класів. Оскільки в базовому класі може бути декілька конструкторів, у класі нащадку необхідно вказати,

який конструктор необхідно викликати. Якщо це не вказано, то викликатиметься конструктор базового класу за замовчуванням.

Який конструктор базового класу викликати – вказується в списку ініціалізації похідного конструктора.

Синтаксис виклику конструктора базового класу:

```
КонструкторБазовогоКласу(формальні параметрі):
КонструкторБазовогоКласу1 (фактичні параметри),
КонструкторБазовогоКласу2(фактичні параметри),…….
КонструкторБазовогоКласуN(фактичні параметри),…….
{тіло конструктора}
```

Наступний приклад демонструє порядок виклику конструкторів та деструкторів.

```
#include<iostream>
using namespace std;
class Base1
{
public:
    Base1() { cout << "Base1\n"; }
    Base1(int a) { cout << "Base1 a\n"; }
    Base1(const Base1& b) { cout << "Base1 copy\n"; }
};

class Base2
{
public:
    Base2() { cout << "Base2\n"; }
    Base2(int a) { cout << "Base2 a\n"; }
    Base2(const Base2& b) { cout << "Base2 copy\n"; }
};

class Derived : public Base1, public Base2
{
public:
    Derived() { cout << "Derived\n"; }
    Derived(int a) :Base1(a) { cout << "Derived a \n"; }
    Derived(int a, int b) :Base1(a), Base2(b)
        { cout << "Derived a b \n"; }
    Derived(const Derived& d) :Base1(d), Base2(d)
        { cout << "Derived copy\n"; }

};

int main()
{
    cout << "Example 1:\n";
    {
        Derived d1;
    }
```

```
    cout << "\nExample 2:\n";
    {
        Derived d2(4);
    }
    cout << "\nExample 3:\n\n";
    {
        Derived d3(2, 3);
    }
    cout << "\nExample 4:\n";
    {
        Derived d4(2, 3);
        Derived d5(d4);
    }
return 0;
}
```

Результат роботи:

```
Example 1:
Base1
Base2
Derived

Example 2:
Base1 a
Base2
Derived a

Example 3:

Base1 a
Base2 a
Derived a b

Example 4:
Base1 a
Base2 a
Derived a b
Base1 copy
Base2 copy
Derived copy
```

Як впливають специфікатори доступу на поля – було розглянуто у підрозділі 12.3. Проте специфікатори доступу також впливають на те, з якими правами перейде розділ базового класу, вони вказуються перед базовим класом:

```
class ім’я_кл_нащадку:public|protected|private базовий_клас {};
```

Якщо специфікатор у попередній конструкції пропущений, то вважається що базовий клас унаслідується з правами доступу *private*.

Якщо базовий клас унаслідується як *public*, то усі розділи базового класу переходять в клас нащадок без змін.

Якщо базовий клас унаслідується як *protected*, то розділи базового класу *public* та *protected* переходять у розділ *protected* класу нащадку, *private* переходять в *private*.

Якщо базовий клас наслідується як *private*, то усі розділи базового класу переходять у *private* класу нащадку. Однак можна вибірково повернути статус деяких елементів базового класу, перевизначивши їх у секції *public* похідного класу.

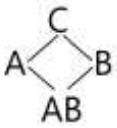
Наприклад, повернення втраченого статусу може виглядати:

```
class Base {
public:
        void f1();
        void f2();
};
class Derived : private Base {
public:
        Base::f1;  // робимо f1() знову відкритим
};
```

Якщо елемент описаний в базовому класі як *private*, його ніяк не можна зробити *public* у похідному класі.

12.15 Віртуальне наслідування

Припустимо, що клас *AB* успадковує від двох класів *A* і *B*, у свою чергу обидва класи *A* і *B* мають базовий клас *C*, як на рисунку:



Нехай клас *C* має деяке поле *data*. При простому наслідуванні у класі *AB* буде два поля *data*, перше прийде через клас *A*, а друге через клас *B*. Якщо створити деякий об’єкт від класу *AB*, то при зверненні до поля *data* необхідно обов’язково вказувати, до якого базового класу воно належить:

```
ім’я_об’єкта.ім’я_класу::ім’я_поля;
```

Наприклад:

```
class C
{
public:
        int data;
};
```

```
class A: public C{};
class B: public C{};
class AB: public A, public B{};
int main()
{
        AB a;
        a.data = 1; //невизначеність
        a.A::data = 1;
        a.B::data = 2;
return 0;
}
```

В наведеному прикладі клас *AB* має два екземпляри класу *C*. Якщо ж необхідно забезпечити, щоб у класі *AB* був тільки один екземпляр класу *C*, необхідно класи *A* та *B* створити через віртуальне наслідування, для цього перед специфікатором доступу вказати ключове слово **virtual**. Наприклад:

```
class C
{
public:
        int data;
};
class A: virtual public C{};
class B: virtual public C{};
class AB: public A, public B{};
int main()
{
        AB a;
        a.data = 1;
return 0;
}
```

В результаті в класі *AB* існує тільки один екземпляр класу *C* і, як результат, тільки одне поле *data*. Класичним прикладом віртуального наслідування є створення класу *iostream*. Він створений від двох класів *istream* та *ostream*, які створюються через віртуальне наслідування від базового класу *ios*.

12.16 Механізм віртуальних методів

Під час роботи з об’єктами одної ієрархії класів існують такі правила:

– покажчик на базовий клас має право зберігати будь-який об’єкт із ієрархії;

– через адресу на базовий клас (покажчик чи посилання) можна передати будь-який об’єкт із ієрархії.

Наприклад:

```
class B {};
class D : public B {};
int main()
{       D d;
        B b;
        B* pb;      // покажчик на базовий клас B
        pb = &b;    //зберігаємо адресу об’єкта B
        pb = &d;    //зберігаємо адресу об’єкта D
        pb = new D; //створюємо динамічний об’єкт D
        delete pb;
return 0;
}
```

Проте загалом не можна в покажчику на нащадок зберігати адресу базового об’єкта, тобто наступне неправильно:

```
    D* pd;
    pd = &b;   // помилка
    pd = &d;   // а так вірно
```

Функція, яка приймає посилання чи покажчик на об’єкт, може прийняти як параметр будь-який об’єкт із ієрархії. Наприклад:

```
class B {};
class D : public B {};
void f1( B*b) {}
void f2(B&b) {}
int main()
{
        D d;
        B b;
        f1(&b);
        f1(&d);
        f2(b);
        f2(d);
return 0;
}
```

Проте під час роботи з об’єктом через покажчик на базовий клас для правильного опрацювання деякого методу необхідно знати, до якого типу належить фактичний об’єкт. Розглянемо це на прикладі.

Створимо наступну ієрархію класів. Як базовий клас створимо клас «монстр» (*Monstr*). Від нього реалізуємо два нащадки – «кролик» (*Rabbit*) та «вовк» (*Wolf*). У цій ієрархії буде спільний інтерфейс (методи). Реалізуємо також деяку функцію *moveByLabrinth()*, яка переміщуватиме (умовно) переданого «монстра» по умовному лабіринту.

```
include<iostream>
using namespace std;
```

```
class Monstr
{
public:
        void move() { cout << "Monstr was moved \n"; }
};

class Rabbit: public Monstr
{
public:
        void move() { cout << "Rabbit was moved \n"; }
};

class Wolf: public Monstr
{
public:
        void move() { cout << "Wolf was moved \n"; }
};
void moveByLabrinth(Monstr& m)
{ m.move(); }


int main()
{
        Rabbit r;
        Wolf w;
        moveByLabrinth(r);
        moveByLabrinth(w);
return 0;
}
```

Результат роботи:

```
Monstr was moved
Monstr was moved
```

У цьому прикладі у функцію було передано об’єкти *Rabbit* і *Wolf* та очікували, що відпрацюють методи відповідних класів, проти в обох випадках відпрацював метод класу *Monstr*. Тобто у функції *moveByLabrinth()* була втрачена інформація про те, який фактично об’єкт їй був переданий, тому вона викликала метод за типом адреси (посилання *Monstr&m*). Щоб виправити цю логічну помилку, в C++ були додані **віртуальні методи**. В попередньому прикладі метод move необхідно оголосити як **віртуальний**, для цього необхідно вказати ключове слово *virtual* перед заголовком методу.

З терміном «віртуальні методи» також зв’язані наступні визначення.

Клас називається **поліморфним,** якщо в ньому є хоча б один віртуальний метод.

Віртуальний метод є **чисто віртуальним**, якщо у нього замість тіла конструкція «=0», наприклад, чисто віртуальний метод:

```
class B
{
        virtual void f() = 0;
};
```

Клас, у якому є хоча б один чисто віртуальний метод, називається **абстрактним**.

Чисто віртуальний метод обов’язково має бути перевизначеним у класі нащадку хоча б знову як чисто віртуальний.

Якщо у базовому класі метод визначений як віртуальний, то він автоматично становиться віртуальним у класі нащадку, навіть якщо ключове слово *virtual* відсутнє.

Створювати об’єкти від абстрактного класу заборонено.

Головне призначення чисто абстрактних методів – це розробка інтерфейсу в базовому класі, коли ще невідомо, як має бути технічно реалізований той чи інший метод.

Так, у попередньому прикладі ми не знаємо, як повинен переміщуватися деякий абстрактний «монстр», проте відомо, як переміщуються кролики та вовки. Тому доцільно у базовому класі *Monstr* метод *move* створити чисто віртуальним:

```
#include<iostream>
using namespace std;

class Monstr
{
public:
        virtual void move() = 0;
};
class Rabbit: public Monstr
{
public:
        virtual void move() { cout << "Rabbit was moved \n"; }
};
class Wolf: public Monstr
{
public:
        virtual void move() { cout << "Wolf was moved \n"; }
};
void moveByLabrinth(Monstr& m) { m.move(); }
int main()
{
        Rabbit r;
        Wolf w;
        moveByLabrinth(r);
        moveByLabrinth(w);
return 0;
}
```

Результат роботи:

```
Rabbit was moved
Wolf was moved
```

Таким чином логічну помилку було виправлено. У функції *moveByLabrinth()* тепер викликаються фактичні методи переданих об’єктів.

Попередні два приклади демонструють відповідно *статичний* та *динамічний поліморфізм*, або ще називають зв’язок першого та другого роду, або раннє та пізнє зв’язування.

Переваги раннього зв’язування – зв’язування виконується на момент компіляції, тобто вже на момент компіляції відомо, який метод викликатиметься – як результат, час виклику звичайного методу швидший порівняно з віртуальним методом.

Переваги пізнього зв’язування (виклику віртуального методу) полягає в тому, що викликаються методи фактичного об’єкта. Пізнє зв’язування виконується в момент роботи програми, тому що на момент компіляції ще невідомо, який фактичний об’єкт буде передано/створено (наприклад, динамічно).

Зауваження: пізнє зв’язування працює тільки у випадку, коли з об’єктом працюють через адресу (покажчик чи посилання).

Для того, щоб з'ясувати, який метод (базового класу чи фактичного об’єкта) викликатиметься, необхідно дотримуватися такого алгоритму:

1) з'ясувати, віртуальний чи не віртуальний метод;
2) якщо метод не віртуальний, то дивимося на тип адреси та викликається метод класу відповідно до типу адреси;
3) якщо метод віртуальний, то дивимося на тип об’єкта, який фактично зберігається за вказаною адресою, та викликається метод фактичного об’єкта.

Наприклад:

```
#include<iostream>
using namespace std;
class B
{
public:
        virtual void f1() { cout << "B::f1\n"; }
        void f2() { cout << "B::f2\n"; }
};
class D:public B
{
public:
        void f1() { cout << "D::f1\n"; }
        void f2() { cout << "D::f2\n"; }
};
```

```
int main()
{       B* p;
        D d;
        p = &d;
        p->f1();
        p->f2();
return 0;
}
```

Результат роботи:

```
D::f1
B::f2
```

У цьому прикладі метод *f1* віртуальний, тому не дивлячись на те, що покажчик *p* належить до типу *B*, метод викликається для фактичного об’єкта *D*. Метод *f2* не віртуальний, тому не дивлячись, що фактичним об’єктом є *D*, викликається метод, що належить до типу покажчика, тобто метод класу *B*.

Розглянемо, як працює механізм віртуальних методів.

Для кожного поліморфного класу, компілятор будує таблицю віртуальних методів (*vtbl*), що зберігає адреси віртуальних методів. Для будь-якого класу в ієрархії наслідування адреса деякого віртуального методу має одне й те саме зміщення в *vtbl*.

Кожен об’єкт поліморфного класу містить прихований покажчик *vptr* на таблицю віртуальних методів класу. Компілятор автоматично вставляє в початок конструктора код, який ініціалізує *vptr* об’єкта. Значення *vptr* покажчика і є тим індикатором приналежності до фактичного класу.

Виклик віртуального методу реалізований через наступний алгоритм:

1) при виклику віртуального методу звертаємося до покажчика *vptr*, за яким переходимо до таблиці *vtbl*;

2) в таблиці *vtbl* за зміщенням, що прив’язане до кожного віртуального методу, знаходимо адресу методу, який необхідно викликати;

3) в пам’яті за знайденою адресою в таблиці *vtbl* знаходимо сам метод, який і викликається.

Як результат, час виклику віртуального методу більший, ніж невіртуального.

12.17 Віртуальні деструктори

Деструктори для поліморфного класу рекомендовано створювати віртуальними. У протилежному випадку звільнення такого динамічного об’єкта може призвести до помилки пам’яті.

Наприклад:

```
#include<iostream>
using namespace std;
class B
{public:
        B() { cout << "B\n"; }
        virtual void f() { cout << "B::f\n"; }
        ~B() { cout << "~B\n"; }
};
class D:public B
{public:
        D() { cout << "D\n"; }
        void f() { cout << "D::f\n"; }
        ~D() { cout << "~D\n"; }
};

int main()
{
        B* p = new D;
        p->f();
        delete p;
return 0;
}
```

Результат роботи:

```
B
D
D::f
~B
```

Як видно з результату роботи, об’єкт *D* був створений правильно (відпрацювали конструктори *B* та *D*), проте конструктор *~D* не був викликаний. Він невіртуальний і оператор *delete* відпрацював за типом покажчика. Якщо деструктор у базовому класі оголосити як віртуальний

```
        virtual~B() { cout<<"~B\n"; },
```

то результат роботи зміниться:

```
B
D
D::f
~D
~B
```

Як результат, динамічний об’єкт звільниться правильно.

12.18 Шаблонні класи

Шаблонні класи (англ. template class) дозволяють створити класи, не прив’язуючись до типу. У свій час велика кількість алгоритмів і контейнерів була додана в C++ у вигляді бібліотеки шаблонів STL (Standard Template Library), через що на тривалий час мова C++ зайняла лідуючі позиції. Такі послідовні контейнери як *vector*, *list*, *deque*, словники *map*, *set* разом з шаблонними алгоритмами дозволяють зберігати будь-які дані та ефективно їх обробляти.

Для того, щоб створити шаблонний клас, необхідно перед заголовком класу вказати ключове слово **template**, а потім у кутових дужках **<>** за допомогою ключового слова **typename** або **class** створити формальний шаблонний тип. Шаблонних типів може бути декілька, вони мають бути відділені комою.

Синтаксис створення шаблонного класу з одним шаблонним типом такий:

```
template <typename (або class)  назва_шабл_типу>
class<назва_класу>
{       // тіло шаблону класу
} <список об’єктів>;
```

Загальний синтаксис створення шаблонного класу з декількома шаблонними типами такий:

```
template <typename (або class) назва_шабл_типу1,
        typename (або class) назва_шабл_типу2, і т.д. >
class [назва_класу]
        {
                // тіло шаблону класу
        }[список_об’єктів];
```

Для того, щоб створити об’єкт від шаблонного класу, необхідно в наступному синтаксису в кутових дужках **<>** вказати фактичні типи, до яких необхідно імплементувати (реалізувати) шаблонний клас:

```
    назва_шаблонного_класу  <фактичний_тип1,  фактичний_тип2  і  так
далі> ім’я_об’єкта(параметри);
```

Наприклад, створимо шаблонний клас, який може зберігати дані будь-якого типу.

```
#include<iostream>
//оголошуємо формальний шаблонний тип MyT

template<typename MyT>
class Data
```

```
{
        MyT data;// створюємо поле від шаблонного типу
public:
        MyT get()const { return data; }
        void set(const MyT& v) { data = v; }
};

int main()
{
  Data<int> d1; //створюємо об’єкт, де поле data отримає тип int
        d1.set(20);
  std::cout << d1.get() << std::endl;
  Data<double> d2;//створюємо об’єкт, де поле data отримає тип double
        d2.set(2.5);
  std::cout << d2.get() << std::endl;
return 0;
}
```

Результат роботи:

```
20
2.5
```

У наведеному прикладі компілятор на етапі розбору шаблонів створить дві реалізації класу: один для типу *int*, другий для типу *double* і дасть кожному із створених класів унікальне ім’я. Тобто компілятор наш код на проміжному етапі замінить на наступний код вже без шаблонів:

```
#include<iostream>

class Dataint
{
        int data;
public:
        int get()const { return data; }
        void set(const int& v) { data = v; }
};
class Datadouble
{
        double data;
public:
        double get()const { return data; }
        void set(const double& v) { data = v; }
};
int main()
{       Dataint d1;
        d1.set(20);
        std::cout << d1.get() << std::endl;
        Datadouble d2;
        d2.set(2.5);
```

```
        std::cout << d2.get() << std::endl;
return 0;
}
```

Таким чином, на базі шаблонного класу буде створена реалізація класу для кожного типу, який був застосований.

Якщо необхідно винести реалізацію методів за межі класу, необхідно дотримуватися такого синтаксису:

```
template <typename (або class) назва_шабл_типу>
назва_шабл_типу назва_шабл_класу<назва_шабл_типу>::ім’я_методу
{ тіло методу }
```

Наприклад, якщо винести у попередньому прикладі реалізацію методів за межі класу, це виглядатиме таким чином:

```
template<typenameMyT>//оголошуємо формальний шаблонний тип MyT
class Data

{       MyTdata;// створюємо поле від шаблонного типу
public:
        MyT get()const;
        void set(const MyT&v);
};
template<typenameMyT>//реалізація методів за межами класу
MyTData<MyT>::get()const
{ return data; }

template<typename MyT>
voidData<MyT>::set(const MyT& v)
{ data = v; }
```

Зауваження:
На етапі розбору шаблонів оголошення класу та реалізація методів мають бути в одному файлі. Тобто не дозволяється виносити реалізацію шаблонних методів в окремий файл **.cpp*, як це рекомендується робити для звичайних (не шаблонних) класів.

Спеціалізація шаблонного класу

У випадку, коли для деякого типу відома більш оптимальна реалізація, ніж загальна, рекомендується виконати спеціалізацію шаблону для конкретного типу. Компілятор завжди надає перевагу спеціалізованій версії класу, а якщо її не існує, то буде виконана імплементація шаблонного класу.

Синтаксис спеціалізації шаблонного класу:

```
template<>
class ім’я_класу_спеціалізації<тип_спеціалізації>
```

```
{
        /* тіло спеціалізованого класу*/
};
```

Наприклад, спеціалізація класу *Data* для типу *int* виглядатиме таким чином:

```
template<>
class Data<int>
{
        int data;
public:

        int get()const { return data; }
        void set(const int&v) { data = v; }
};
```

Шаблонний цілий тип

Які шаблонний тип може бути цілий тип, при створенні об’єкта від такого класу необхідно вказати цілу константу.

Приклад синтаксису шаблонного цілого типу:

```
template<typename назва_шабл_типу, int назва_константи>
class<назва_класу>
{
        // тіло шаблону класу
} <список об’єктів>;
```

Наприклад, створимо шаблонний клас, який зберігає звичайний масив довільного типу. Розмір масиву задається шаблонним параметром цілого типу.

```
#include<iostream>
template<typename MyT, int size>
class Arr
{
        MyT mas[size];
        const int n;
public:
        Arr():n(size)
        {
                for (int i = 0; i < n; ++i)
                     mas[i] = i * i / 2.0;
        }
        void print()const
        {
                for (int i = 0; i < n; ++i)
                     std::cout << mas[i] << " ";
        }
};
```

```
int main()
{
        Arr<int, 5> a1; a1.print();
        std::cout << "\n";
        Arr<double, 8> a2; a2.print();
return 0;
}
```

Результати роботи:

```
0 0 2 4 8
0 0.5 2 4.5 8 12.5 18 24.5
```

В результаті було створено дві різні реалізації шаблонного класу, перша – для типу *int* і розмір масиву дорівнює 5, другий клас для типу *double* і розміру масиву 8.

Наслідування від шаблонного класу

При створенні нащадку від шаблонного класу обов’язково необхідно вказувати, для якого типу необхідно зробити реалізацію.

Синтаксис:

```
class ім’я_класу_нащадка :
public ім’я_базового_шабл_класу<тип_реалізації>
{
        /*синтаксис створення конструктора */
        ім’я_класу_нащадка(параметри):
ім’я_базового_шабл_класу<тип_реалізації>
        {
        /*тіло конструктора */
        }
};
```

Наприклад, створимо шаблонний клас з конструктором і нащадка для типу *int*:

```
// шаблонний клас
template<typenameMyT>
class Data
{
        MyT data;
public:
        Data(const MyT&d) :data(d) {}
};
// нащадок від шаблонного класу для типу int
class DataInt : public Data<int>
{
public:
        DataInt(intd) :Data<int>(d) {}
};
```

Якщо необхідно створити нащадка, який знову буде шаблонним класом, це виглядатиме таким чином:

```
#include<iostream>
#include<string>
// шаблонний клас
template<typename MyT>
class Data
{
protected:
        MyT data;
public:
        Data(const MyT& d) :data(d) {}
};
// шаблонний нащадок від шаблонного класу
template<typename MyT>
class NewData: public Data<MyT>
{
public:
        NewData(const MyT & d) :Data<MyT>(d) {}
        void print()const { std::cout << this->data << std::endl; }
};
int main()
{
        NewData<std::string> d("information");
        d.print();
return 0;
}
```

Результат роботи:

```
information
```

Зауваження:
При зверненні до поля шаблонного базового класу, необхідно завжди вказувати *this* (у прикладі це *this->data*) перед полем.

12.19 Запитання та завдання

1. Які особливості використання ООП?
2. Що таке інкапсуляція, для чого вона використовується?
3. Охарактеризуйте поняття «успадкування».
4. Що таке поліморфізм, віртуальні методи?
5. Для чого використовуються конструктори та деструктори?
6. Охарактеризуйте правила запису функцій членів класів.

Завдання.

Набуття практичних навичок роботи з класами під час розв’язання завдань розділу вправ «Класи».

ВПРАВИ

1. ОБЧИСЛЕННЯ МАТЕМАТИЧНИХ ВИРАЗІВ

1. Обчислити висоти $\mathbf{h_a}$, $\mathbf{h_b}$, $\mathbf{h_c}$ трикутника **ABC**, якщо значення сторін **a**, **b** і **c** вибрано довільно.

Довідка:

$$\mathbf{h_a} = \frac{2}{a}\sqrt{p(p-a)(p-b)(p-c)},\ p = \frac{a+b+c}{2}.$$

Відповідно знаходяться висоти $\mathbf{h_b}$ та $\mathbf{h_c}$.

2. Обчислити об'єм кульового сегмента та об'єм кульового сектора за формулами:

$$V = \pi H^2\left(R - \frac{H}{3}\right),\ V = \frac{2}{3}\pi R^2 H.$$

Значення радіуса кулі **R** та висоти **H** кульового сегмента або сегментної частини сектора задати довільно.

3.Обчислити

$a = \frac{2\cos\left(x-\frac{\pi}{6}\right)b}{\frac{1}{2}+\sin^2(y)}$, де $b = 1 + \frac{z^5}{3+\frac{z^2}{5}}$, якщо **x** = 1.45; **y** = -1.22; **z** = 3.5.

4. Обчислити

$s = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!}$, $z = \sin(x^3) + \cos^2(y)$, якщо **x** = 1.2, **y** = -0.8.

Довідка: **n! = 1 · 2 · 3 … · n**.

5. Обчислити площу поверхні зрізаного конуса і його об'єм за формулами:

$$S = \pi(R + r)L + \pi R^2 + \pi r^2,\ L = \sqrt{h^2 + (R - r)},\ V = \frac{\pi}{3}(R^2 + r^2 + Rr)h.$$

Значення радіусів основ зрізаного конуса **R** і **r**, його висоту **H** та довжину твірної **L** вибрати довільно.

6. Обчислити

$$y = e^{-bt}\sin(at + b) - \sqrt{|bt + a|},\ s = b\cdot\sin(at^2\cdot\cos(2t)) - 1,$$

якщо **a** = -0.5; **b** = 1.7; **t** = 0.44.

7. Обчислити

$$P = \sqrt{x^2 + b} - b^2\frac{\sin^3(x+a)}{x\cdot y},\ \text{де}\ y = \cos^2(x^3) - \frac{x}{\sqrt{a^2+b^2}},$$

якщо **a** = 1.5; **b** = 15.5; **x** = -2.9.

8. Обчислити

$$\mathbf{S} = \frac{\mathbf{x^3\,tg^2(x+b)^2+a}}{\sqrt{\mathbf{x+b}}}\mathbf{c},\ \text{де}\ \mathbf{c} = \frac{\mathbf{bx^3-a}}{\mathbf{e^{ax}-1}},\ \text{якщо}\ \mathbf{a} = 16.5;\ \mathbf{b} = 3.4;\ \mathbf{x} = 0.61.$$

9. Обчислити

$$\boldsymbol{\gamma} = \left|\mathbf{x}^{\mathbf{y/x}} - \sqrt[3]{\mathbf{y/x}}\right|,\ \boldsymbol{\varphi} = (\mathbf{x}-\mathbf{y})\frac{\mathbf{y}-\frac{\mathbf{z}}{\mathbf{y-x}}}{\mathbf{1+(y-x)^2}},\ \text{якщо}\ \mathbf{x} = 1.82;\ \mathbf{y} = 18.5;\ \mathbf{z} = -3.4.$$

10. Обчислити

$$\mathbf{R} = \frac{\mathbf{x^2(x+1)}}{\mathbf{b}} - \mathbf{sin^2(x-a)},\ \mathbf{S} = \sqrt{\mathbf{x}\frac{\mathbf{b}}{\mathbf{a}}} + \mathbf{|cos(x+b)^3|},$$

якщо $\mathbf{a} = 0.7$; $\mathbf{b} = 0.05$; $\mathbf{x} = 0.43$.

11. Обчислити

$$\mathbf{a} = \frac{\sqrt{|\mathbf{x-1}|}-\sqrt{\mathbf{y}}}{\mathbf{1}+\frac{\mathbf{x^2}}{\mathbf{2!}}+\frac{\mathbf{x^2}}{\mathbf{4!}}},\ \mathbf{b} = \mathbf{x(arctg(z) + e^{x+3})},$$

якщо значення **x**, **y**, **z** вибрані довільно.

12. Обчислити

$$\mathbf{p} = \frac{\mathbf{a}}{\mathbf{b}},\ \text{де}\ \mathbf{a} = (\mathbf{1}-\mathbf{y})\frac{\frac{\mathbf{(x+y)^2}}{\mathbf{(x+4)^3}}}{\mathbf{e^{-(x-2)}+(x^3+4)}},\ \mathbf{b} = \frac{\mathbf{1+cos(y-2)}}{\mathbf{x+sin^2(y-2)}},\ \text{якщо}\ \mathbf{x} = 1.25;\ \mathbf{y} = 0.93.$$

13. Обчислити

$$\mathbf{P} = \left|\frac{\mathbf{sin^3(ax^3+by^2-ab)}}{\mathbf{\sqrt[3]{(ax^3+by^2-a)^2}+\pi}}\right| + \mathbf{tg(a{\cdot}x^3 + b{\cdot}y^2 - a{\cdot}b)},$$

якщо $\mathbf{x} = 0.25$; $\mathbf{y} = 1.31$; $\mathbf{a} = 3.5$; $\mathbf{b} = 0.9$.

14. Обчислити

$$\mathbf{P} = \frac{\mathbf{1+sin^2(x+1)}}{\mathbf{2+\left|x-\frac{2x^3}{1+x^2y^3}\right|}} + \mathbf{x^4},\ \mathbf{Q} = \mathbf{cos^2(arctg(\frac{1}{z}))},\ \text{якщо}\ \mathbf{x} = 0.25;\ \mathbf{y} = 0.79;\ \mathbf{z} = 0.81.$$

15. Обчислити

$$\mathbf{Y} = \mathbf{b^5\,tg^2(x)} - \frac{\mathbf{a}}{\mathbf{\sin^2\left(\frac{x}{a}\right)}},\ \mathbf{Z} = \mathbf{a{\cdot}e^{-\sqrt{3}}\cos\left(\frac{bx}{a}\right)},\ \text{якщо}\ \mathbf{a} = 3.2;\ \mathbf{b} = 17.5;\ \mathbf{x} = -4.8.$$

16. Обчислити

$$\mathbf{K} = \mathbf{ln(a + x^3)} + \mathbf{sin^4\left(\frac{x}{b}\right)},\ \mathbf{M} = \mathbf{e^{-cx}}\frac{\mathbf{x-\sqrt[3]{x+a}}}{\mathbf{x-\sqrt{|x-b|}}},$$

якщо $\mathbf{a} = 10.2$; $\mathbf{b} = 9.3$; $\mathbf{x} = 2.4$; $\mathbf{c} = 0.5$.

17. Обчислити

$$\mathbf{Y} = \frac{\mathbf{a^{2x}+b^{-x}x{\cdot}cos(a+b)}}{\mathbf{|x-1|}},\ \mathbf{R} = \sqrt{\mathbf{x^2+b}} - \mathbf{b^2\,sin^3\left(\frac{x+a}{x}\right)},$$

якщо $\mathbf{a} = 0.3$; $\mathbf{b} = 0.9$; $\mathbf{x} = 0.53$.

18. Обчислити

$\alpha = \frac{a^x + b^{-x}\sin(a-b)}{\sqrt{|a-b|}}$, $\beta = ae^{\sqrt{a}}\cos\left(\frac{x}{a}\right)$, якщо **a** = 0.5; **b** = 2.9; **x** = 0.3.

19. Обчислити

$$\alpha = \sqrt{|ax^2\sin(2x) - e^{-2x}(x+b)|},\ \varphi = \frac{1}{\cos^2(x^3)} - \frac{x}{\sqrt[3]{a^2+b^2}},$$

якщо **a** = 0.5; **b** = 3.1; **x** = 1.4.

20. Обчислити

$$U = \frac{a^3 + e^{-x}\cos(bx)}{bx - e^{-x}\sin(bx+1)},\ F = e^{2x}\ln(a+x) - e^{3x}\ln|x+b|,$$

якщо **a** = 0.5; **b** = 2.9; **x** = 0.3.

21. Обчислити

$a = y + \frac{x}{y^3 + \left|\frac{x^2}{y + \sqrt[3]{x^2}}\right|}$, $b = (1 + tg^2(\frac{x}{2}))\ln(x)$, якщо **x** = 1.23; **y** = 0.79.

22. Обчислити

$P = \frac{3 + e^{y-1}}{1 + x^2|y - tg(z)|}$, $S = 1 + y - \frac{(y-x)^2}{2} + \frac{|y-x|}{3}$, якщо **x** = 10.3; **y** = 4.93; **z** = 0.4.

23. Обчислити

$A = \frac{z\cos\left(x - \frac{\pi}{6}\right)}{\frac{1}{2} + \sin^3(x)}$, $B = 1 + \frac{z^3}{3 + \frac{z^2}{5}}$, якщо **x** = 1.2; **z** = 3.5.

24. Обчислити

$$P = \frac{\sin^3(|cx^2 + by^2|)}{\sqrt[3]{|cx^3 + by^2|}} + tg(cx^3 + by^2),\ S = \cos^2(x^3) - x\sqrt{(x-b)^2},$$

якщо **c** = 0.5; **b** = -0.5; **x** = 0.61; **y** = 1.2.

25. Обчислити

$A = \frac{\cos(x^3 - \pi/2)}{1 + \lg(y-2)} + \sqrt[3]{y-2}$, $B = \frac{a^2x + y^{-x}\cos(a+x)x}{|x-1|}$, якщо **x** = 0.92; **y** = 5.3; **a** = 0.25.

2. ЦИКЛІЧНІ ОБЧИСЛЮВАЛЬНІ ПРОЦЕСИ

1. Знайти суму всіх значень функції $y = x^i/i$, якщо $i \in [1; 10]$, $h_i = 1$ та $x \in [0; 1]$, $h_x = 0.1$.

2. Обчислити функцію **y** та знайти добуток її додатних значень:

$$y = \begin{cases} ax + b\cos(x) + c, & \text{якщо } x \le 0.5; \\ bx^2 + c\sin(2x), & \text{для інших значень } x, \end{cases}$$

де **a** = 0.75; **b** = 1.19; **c** = -2.5; $x \in [0; 2]$, $h_x = 0.1$.

3. Обчислити функцію **z** та підрахувати кількість її від'ємних значень:

$$z = \begin{cases} a^x + b^y, & \text{якщо } x + y < 1; \\ ax^2 + \ln(bxy), & \text{якщо } x + y \geq 1, \end{cases}$$

де **a** = 1; **b** = 2; **x** ∈ [1; 2], h_x = 0.1; **y** ∈ [1; 2], h_y = 0.2.

4. Обчислити функцію **y** та підрахувати суму її перших трьох значень:

$$y = \begin{cases} \sin(|ax + b|), & \text{якщо } x < \frac{a}{b}; \\ \cos(|ax - b|), & \text{якщо } x \geq \frac{a}{b}, \end{cases}$$

де **a** = 5; **b** = 3; **x** ∈ [2; 8], h_x = 1.

5. Обчислити функцію **y** та знайти її найбільше значення:

$$y = \frac{4ax^2 + 37x + b}{a - 0.5},$$

якщо **b** = 1.2; **a** ∈ [1; 2], h_a = 0.5; **x** ∈ [0; 1], h_x = 0.2.

6. Обчислити функцію **y** та знайти середнє арифметичне її значень:

$$y = \begin{cases} x^4 + 2x^2 \cos(x), & \text{якщо } |x| \leq 3; \\ \frac{\sin(x + 0.4)^2}{4x^2}, & \text{якщо } |x| > 3, \end{cases}$$ де **x** ∈ [-5; 5], h_x = 1.

7. Обчислити функцію **z** та підрахувати кількість її значень, що перевищують число, задане користувачем:

$$z = \begin{cases} \frac{x}{15a^2}, & \text{якщо } x^2 + y^2 \leq a^2; \\ x^2 + e^x, & \text{якщо } x^2 + y^2 > a^2, \end{cases}$$

де **a** = 5.3; **y** = 1.1; **x** ∈ [-3; 3], h_x = 0.5.

8. Обчислити функцію **y** та для кожного значення параметра **b** знайти суму її від'ємних значень:

$$y = e^{\frac{x}{2}} b^x (\cos(z) + \sin(z)), \text{ де } z = \frac{\sqrt{4b + x^2}}{2},$$

якщо **b** = 2; 4; 6; 8; **x** ∈ [1.2; 2], h_x = 0.2.

9. Обчислити добуток **y** згідно з умовою, якщо **x** ∈ [0; 0.5], h_x = 0.1:

$$y = \prod_{k=1}^{7} \left(1 + \frac{\sin(kx)}{k!}\right),$$

Довідка: **k! = 1 · 2 · 3... · k**.

10. Обчислити функцію **y** та знайти середнє геометричне отриманих значень, де **x** ∈ [-2; 4], h_x = 0.5:

$$y = \begin{cases} 1 - \cos(x) + \sqrt[3]{x}, & \text{якщо } x \geq 0; \\ \frac{\cos^2(x) - \sin(x)}{x^3}, & \text{якщо } x < 0, \end{cases}$$

11. Обчислити функцію **y** та знайти кількість її від'ємних значень для кожного значення параметра **c,** де **a** = 2; **b** = 3 ; **c** = 1; 3; 5; 7; **x** ∈ n[0.1; 5], $\mathbf{h_x}$ = 0.1:

$$\mathbf{y} = \begin{cases} \mathbf{a^{b-x} + c}, & \text{якщо } \mathbf{x} < 2; \\ \mathbf{b^{c-x} + a}, & \text{якщо } \mathbf{2 \le x \le 4}; \\ \mathbf{c^{a-x} + b}, & \text{якщо } \mathbf{x} > 4, \end{cases}$$

12. Обчислити площі **S** геометричних фігур, де параметри **a**, **b**, **h**, **R** задані довільно:

$$\mathbf{S} = \begin{cases} \mathbf{ab}, & \text{якщо } \mathbf{n = 1}; \\ \dfrac{\mathbf{ah}}{\mathbf{2}}, & \text{якщо } \mathbf{n = 2}; \\ \dfrac{\mathbf{(a+b)h}}{\mathbf{2}}, & \text{якщо } \mathbf{n = 3}; \\ \mathbf{\pi R^2}, & \text{якщо } \mathbf{n = 4}. \end{cases}$$

13. Обчислити функцію **z** та знайти для кожного парного значення параметра **y** кількість додатних значень цієї функції, де **x** ∈ [1; 10], $\mathbf{h_x}$ = 2; **y** ∈ [-4; 3], $\mathbf{h_y}$ = 1:

$$\mathbf{z} = \begin{cases} \dfrac{\mathbf{x^2}}{\mathbf{(x-5)^3}}, & \text{якщо } \mathbf{x > y}; \\ \dfrac{\mathbf{(x-2)^3}}{\mathbf{y(x-5)^4}}, & \text{якщо } \mathbf{x \le y}. \end{cases}$$

14. Обчислити значення функції **z** при зміні кожного з параметрів **x**, **y**, **b,** де **a** = 1.7; **b** = 2; 4; 6; 8; **x** ∈ [-1; 1], $\mathbf{h_x}$ = 0.9; **y** ∈ [-2; 2], $\mathbf{h_y}$ = 1:

$$\mathbf{z} = \begin{cases} \mathbf{\ln(\sqrt{ax^2 + by^3})}, & \text{якщо } \mathbf{xy > 0}; \\ \mathbf{a^x + b^y}, & \text{якщо } \mathbf{xy \le 0}. \end{cases}$$

15. Обчислити суму згідно з умовою для кожного значення параметра **a,** де $\mathbf{h_x}$ = 0.5; **a** = 1, 2, 3, 4, 5:

$$\mathbf{y} = \sum_{\mathbf{x=1}}^{\mathbf{10}} \mathbf{x\sqrt{ax^3}}.$$

16. Підрахувати, скільки разів функція $\mathbf{y = a \cdot \cos(x) + b \cdot e^{-x}}$ приймає додатні значення, якщо **x**∈[0.3; 5], $\mathbf{h_x}$ = 0.1; **a**∈[2; 8],$\mathbf{h_a}$ = 2; **b**∈[-0.1; 2.2], $\mathbf{h_b}$ = 0.1.

17. Обчислити значення функції **y** згідно з умовою, де **x** ∈ [0.5; 3.1], $\mathbf{h_x}$ = 0.2; **a** = 0.1, $\mathbf{h_a}$ = 0.2; параметри **x** і **a** змінюються одночасно:

$$\mathbf{y} = \begin{cases} \mathbf{\cos(ax + 2)}, & \text{якщо } \mathbf{x > 2}; \\ \mathbf{tg(|x - 2a|)}, & \text{якщо } \mathbf{x \le 2}. \end{cases}$$

18. Обчислити функцію **y** та підрахувати добуток її перших шести значень:

$$y = \begin{cases} 1 - \sin(x), & \text{якщо } 5 < x < 10; \\ \frac{1}{2(1+\cos(x))}, & \text{якщо } 10 < x < 15; \\ 1/3(\ln(x)), & \text{якщо } 15 < x < 20; \\ ctg^2(x), & \text{якщо } 20 < x < 25, \end{cases}$$ де $x \in [5; 25]$, $h_x = 0.5$.

19. Обчислити значення функції **y** згідно з умовою:

$$y = \begin{cases} n(n-2)(n-1), & \text{якщо } n = 6; \\ 2^{28-n}, & \text{якщо } n = 10; \\ (n-10)!, & \text{якщо } n = 18, \end{cases}$$ де $n \in [0; 20]$, $h_n = 2$.

20. Обчислити значення **y** згідно з умовою:

$$y = \prod_{x=1}^{4} \sum_{z=2}^{10} (\sqrt[5]{x} + \lg(z)),$$ де $h_z = 2$; $h_x = 1$.

21. Обчислити значення функції **y** згідно з умовою, де $x \in [0; 8]$, $h_x = 2$; параметри **a** та **b** взяти довільно:

$$y = \begin{cases} a + bx^3, & \text{якщо } 1 \le x < 2; \\ a\sin^3(x+b), & \text{якщо } 2 \le x < 3; \\ \sqrt{|a + bx^3|}, & \text{якщо } 3 \le x < 4; \\ a \cdot \lg(|b + 2ax|), & \text{якщо } 4 \le x < 5; \\ e^{a\sin(x)}, & \text{якщо } 5 \le x < 6, \end{cases}$$.

22. Ввести координати десяти точок на площині та визначити, яка чверть координатної площини містить найбільшу кількість заданих точок.

23. Ввести п’ять наборів значень сторін трикутника **a**, **b**, **c** і визначити, для яких сторін висота h_a буде найменшою.

Довідка: сторона визначається за формулою

$$h_a = \frac{2}{a}\sqrt{p(p-a)(p-b)(p-c)},\ p = \frac{a+b+c}{2}.$$

24. Ввести координати п’яти точок і визначити, яка з них потрапить у коло радіуса **R** і координатами центра (**a**, **b**).

Довідка: рівняння кола має вигляд $(x - a)^2 + (y - b)^2 = R^2$.

25. Ввести шість шестирозрядних цілих чисел і знайти, для яких з них сума «лівих» трьох цифр більше суми «правих» трьох.

3. ОДНОВИМІРНІ ТА ДВОВИМІРНІ МАСИВИ

1. За вказівкою користувача обчислити кількість та суму додатних елементів або додаток від'ємних непарних елементів масиву **m[15]**.

2. За вказівкою користувача з додатних або від'ємних елементів матриці **mt[4][5]** утворити одновимірний масив, елементи якого розмістити в зворотній послідовності.

3. За вказівкою користувача обчислити або добуток та суму елементів розташованих на парних місцях масиву **x[15],** або непарні елементи масиву відсортувати за зменшенням.

4. За вказівкою користувача відсортувати або заданий рядок матриці **xm [3][5]** за зростанням, або підрахувати стовпці, в яких максимальний елемент перевищує число 9.

5. За вказівкою користувача відсортувати або всі стовпці матриці **xm[3][5],** або заданий рядок за зменшенням додатних елементів.

6. За вказівкою користувача з елементів масиву **a[15]** сформувати масив або від'ємних, або додатних елементів.

7. За вказівкою користувача знайти або мінімальний елемент і його індекс серед від'ємних елементів масиву **x[15],** або мінімальний елемент і його індекс серед парних елементів цього масиву.

8. За вказівкою користувача розділити всі елементи матриці **a[3][5]** або на суму її додатних елементів, або на кількість непарних елементів.

9. За вказівкою користувача або здійснити додавання та віднімання матриць **a[2][3]** і **b[2][3]**, або знайти добуток парних елементів матриць.

10. За вказівкою викладача або поміняти місцями задані рядки матриці **mt[4][5]**, або здійснити сортування заданих стовпців цієї матриці.

11. За вказівкою користувача або обчислити суму елементів матриці **m[4][4],** що знаходяться вище та нижче головної діагоналі, або знайти додаток елементів кожного стовпця матриці.

12. За вказівкою користувача або обчислити суму та добуток від'ємних елементів одновимірного масиву **x[20]**, або знайти максимальний елемент серед парних елементів цього масиву.

13. За вказівкою користувача визначити або максимальний елемент в кожному рядку матриці **a[3][5]**, або мінімальний елемент серед додатних елементів у кожному стовпці матриці.

14. За вказівкою користувача визначити або кількість від'ємних елементів головної діагоналі матриці **m[5][5]**, або суму елементів, що знаходяться вище головної діагоналі.

15. Вставити в одновимірний масив **m[10]** новий елемент у місце, що зазначене користувачем, не порушуючи розташування інших елементів.

16. За вказівкою користувача розділити всі елементи масиву **x[4][3]** або на кількість додатних, або на суму максимального та мінімального елементів масиву.

17. Знайти за вказівкою користувача індекс або максимального, або мінімального елемента масиву **m[15]** та записати всі елементи, що розташовані після цього елемента в новий масив.

18. Всі непарні та від'ємні елементи матриці **m[4][5]** розділити або на її перший, або деякий інший елемент за вказівкою користувача.

19. Обчислити або суму елементів кожного рядка матриці **v[4][5]**, або кількість додатних елементів стовпців цієї матриці за вказівкою користувача.

20. За вказівкою користувача або всі непарні елементи матриці **c[4][4]** переписати у масив **c1**, або додатні елементи трьох рядків — у масив **c2.**

21. У відсортований за зростанням масив **b[10]** вставити у зазначене користувачем місце новий елемент, не порушуючи сортування.

22. За вказівкою користувача визначити кількість парних елементів матриці **a[5][5]**, що знаходяться вище головної або нижче побічної діагоналі.

23. За вказівкою користувача обчислити або суму від'ємних елементів і добуток додатних, або кількість елементів, що дорівнюють заданому числу в масиві **x[20]**.

24. За вказівкою користувача всі елементи масиву **c[15]** розділити або на його максимальний елемент, або на мінімальний додатний елемент.

25. За вказівкою користувача відсортувати або всі стовпці матриці **p[4][5]** за зростанням значень елементів, або задані рядки матриці за зменшенням значень їх елементів.

4. РЯДКИ

1. Ввести речення та підрахувати і вивести слова, що починаються на сполучення літер «св».

2. У наведеному тексті вивести слова, що починаються та закінчуються на однакові літери.

3. Ввести речення та визначити і вивести найкоротше слово.

4. З уведеного списку прізвищ вивести прізвища, що починаються та закінчуються на задані користувачем літери.

5. Визначити, скільки разів у введеному реченні повторюється літера «о».

6. У введеному тексті видалити між словами всі символи, що не є літерами.

7. Ввести деяку кількість слів та визначити три найдовших слова.

8. Ввести список прізвищ, що розташовані у довільному порядку, та відсортувати його за алфавітом.

9. Ввести будь-яке речення та визначити, скільки разів повторюється в ньому задане слово.

10. Введений текст розбити на слова і вивести їх за алфавітом.

11. З введеного списку прізвищ вивести ті, що мають дві голосні літери.

12. Ввести речення та вивести слова, у яких немає сполучення літер «про» та літери «р».

13. Ввести текст та підрахувати слова, що збігаються із заданим словом.

14. У наведеному тексті замінити слово «урок» на «досвід»: «Помилка одного — урок іншому».

15. З уведеного тексту вивести слова із закінченням «ти»: «Існує лише один спосіб стати добрим співрозмовником — вміти слухати».

16. З уведеного списку вивести прізвища із заданою кількістю літер.

17. Вивести слово, що складене з перших літер слів уведеного речення.

18. Вивести частину наведеного тексту, що розташована між словами «укладено» та «є»: «У кожній природничій науці укладено стільки істини, скільки в ній є математики».

19. Ввести невелику програму, написану мовою C++. Визначити, скільки в ній операторів і скільки разів повторюється оператор *for*.

20. Підрахувати, скільки слів у наведеному реченні містять більше ніж п’ять символів та вивести ці слова: «Істина, виражена словами, є могутня сила у житті людей».

5. СТРУКТУРИ

1. Враховуючи відомості про заборгованості групи студентів, вивести за вказівкою користувача або повідомлення про студентів — харків’ян, які мають заборгованості та їхню кількість, або повідомлення про іногородніх студентів — боржників.

2. Враховуючи відомості про студентів вашої групи: прізвище, ініціали, місце та дату народження, вивести за вказівкою користувача або повідомлення про студентів, які народилися у заданому місяці, або список студентів за зростанням року народження.

3. Враховуючи відомості про студентів вашої групи: прізвище, ініціали, місце проживання, навчання на підготовчому відділенні — денна, недільна, заочна форма навчання, вивести за вказівкою користувача або повідомлення про студентів, які закінчили підготовче відділення заданої форми навчання, або список студентів, які взагалі не навчались там.

4. Ввести інформацію про працівників виробничого підрозділу такого змісту: прізвище, ініціали, стать, рік народження та спеціальність. Вивести за вказівкою користувача повідомлення або про працівників жіночої статі заданої спеціальності, або про працівників чоловічої статі заданого року народження.

5. Ввести інформацію про студентів, що проживають у гуртожитку, такого змісту: прізвище, стать та номер кімнати. Вивести за вказівкою користувача або номера кімнат, у яких проживають юнаки, або прізвища дівчат, котрі мешкають в одній кімнаті.

6. Ввести відомість успішності студентів з трьох предметів. Вивести за вказівкою користувача або список студентів з їхнім середнім балом, або дані про студента за заданим прізвищем та ініціалами.

7. Ввести інформацію про працівників виробничого підрозділу такого змісту: прізвище, ініціали, вік та спеціальність. Вивести за вказівкою користувача або середній вік працівників заданої спеціальності, або список працівників, прізвища яких починаються на задану літеру.

8. Ввести інформацію про результати складання іспиту з дисципліни «Програмування» студентами деякої групи. Вивести за вказівкою користувача список студентів, що отримали добрі оцінки, або список студентів за алфавітом, що не склали іспит взагалі.

9. Ввести список студентів з інформацією про їхні оцінки з трьох предметів. За вказівкою користувача вивести список або за зростанням середнього балу успішності, або за спаданням середнього балу.

10. Ввести список викладачів кафедри з такою інформацією: прізвище та ініціали, стать, посада, назва дисципліни, яку викладає. Вивести за вказівкою користувача або список викладачів — жінок заданої посади, або повідомлення про викладачів — чоловіків.

11. Ввести список літератури з програмування з такою інформацією: прізвища авторів, назва книги та видавництва, рік і місце видання.

Вивести за вказівкою користувача або повідомлення про літературу, що вийшла у заданому році, або назву видань заданих авторів.

12. Ввести список студентів, які проживають у гуртожитку, враховуючи прізвище, ініціали, рік народження, номер гуртожитку та кімнати. Вивести за вказівкою користувача або повідомлення про студентів, які проживають у заданому гуртожитку та народились у заданому році, або прізвища студентів, які проживають у кімнатах із заданим номером у двох заданих гуртожитках.

13. Ввести інформацію про групу студентів: прізвище та ініціали, місце народження, захоплення, стать. Вивести за вказівкою користувача повідомлення про студенток, які захоплюються спортом, або список юнаків, які люблять програмувати та народилися у Харкові.

14. Ввести список, у якому зазначені прізвища, ініціали, адреса та номери телефонів абонентів. Вивести за вказівкою користувача або прізвище абонента за заданою адресою та номером телефону, або повідомлення про абонентів, що мають задане прізвище та ініціали.

15. Ввести список, який містить прізвище, ініціали і рік народження працівників окремого підрозділу. Вивести за вказівкою користувача повідомлення про працівників заданого року народження, або список прізвищ за зростанням року народження.

16. Ввести розклад занять із зазначенням дня тижня, назви предмета, виду заняття (лекції, практичні, лабораторні) та часу проведення. Вивести за вказівкою користувача дні тижня, коли із заданого предмета проводяться лекції, або повідомлення про заняття в заданий день тижня.

17. Ввести список літературних джерел з дисциплін, що викладаються у поточному семестрі. За вказівкою користувача вивести повідомлення або про літературу з заданої дисципліни, або про назви книг заданого року видання.

18. Ввести в комп'ютер інформацію про захворювання співробітників: прізвище та ініціали, рік народження, захворювання, тривалість хвороби. Вивести за вказівкою користувача або прізвище працівника, який хворів найдовше, або список працівників за зростанням тривалості хвороби.

19. Ввести список спортсменів, що складається з прізвища, статі, року народження та виду спорту. Вивести за вказівкою користувача список легкоатлетів або список жінок-гімнасток.

20. Ввести в комп'ютер інформацію про асортимент продовольчих товарів у магазині такого змісту: назва магазину, код товару, кількість цього товару, ціна за кілограм. За вказівкою користувача або відсортувати магазини за спаданням оптової ціни заданого товару, або вивести повідомлення, в яких магазинах є потрібний товар.

21. Ввести в комп'ютер таку інформацію про потяги: номер потягу, назву напрямку руху, час прибуття. Вивести за вказівкою користувача або повідомлення про потяги, що прибувають у заданий час, або про кількість потягів заданого користувачем напрямку руху.

22. Ввести в комп'ютер інформацію про книги з програмування мовою C++. На екран за вказівкою користувача вивести або повідомлення про книги вітчизняних авторів, або список книг закордонних авторів, розташованих за спаданням кількості сторінок.

23. Ввести в комп'ютер таку інформацію про автомобілі: прізвище та ініціали власника, модель автомобіля, рік його випуску та потужність. Вивести за вказівкою користувача марки автомобілів, рік випуску та потужність яких співпадають з введеними в рядку запиту, або прізвища власників автомобілів за алфавітом з відповідними повідомленнями про їхні автомобілі.

24. Ввести в комп'ютер інформацію про користувачів бібліотеки: прізвище та ініціали читача, назву кафедри, кількість книг на абонементі. Вивести за вказівкою користувача або повідомлення про читача, що має найбільшу кількість книг, або список працівників заданої кафедри за зростанням кількості книг.

25. Ввести інформацію про студентів групи: прізвище та ініціали, середні бали за два семестри. Вивести за вказівкою користувача або повідомлення про студентів, що мають однаковий середній бал за обидва семестри, або список студентів за спаданням їхнього середнього балу в другому семестрі.

6. ФУНКЦІЇ

1. Розробити функцію, що знаходить середнє геометричне всіх від'ємних елементів масиву та використати для трьох масивів різної довжини.

2. Розробити функцію видалення підрядка в **n** символів з **k**-ї позиції введеного рядка та використати її для обробки декількох рядків тексту.

3. Скласти функцію, що обчислює середнє арифметичне та середнє геометричне додатних елементів матриці. Застосувати розроблену функцію для обробки трьох двовимірних масивів.

4. Обчислити площу багатокутника за допомогою функції, що визначає площу за координатами його вершин.

5. Розробити функцію визначення середнього значення парних елементів матриці та мінімального серед непарних елементів та з її допомогою обробити три двовимірних масиви.

6. Скласти функцію знаходження середнього арифметичного значення елементів масиву, розташованих на непарних місцях, та застосувати її для обробки декількох одновимірних масивів.

7. Розробити функцію, яка здійснює перестановку максимального від'ємного та мінімального додатного елементів одновимірного масиву, та застосувати її для обробки чотирьох масивів різної довжини.

8. Розробити функцію сортування стовпців двовимірних масивів за зростанням і застосувати її для обробки декількох матриць довільного розміру.

9. Розробити функцію, яка записує від'ємні непарні елементи довільної матриці в одновимірний масив, та застосувати її для обробки матриць трьох матриць **A[5][4]**, **B[3][2]**, **C[4][4]**.

10. Розробити функцію, що обчислює суму діагональних елементів квадратної матриці, та за її допомогою знайти суми діагональних елементів трьох матриць.

11. Скласти функцію визначення суми та кількості елементів, що розташовані між мінімальним і максимальним елементами матриці, та застосувати її для обробки трьох довільних двовимірних масивів.

12. Розробити функцію, що здійснює підрахунок кількості непарних та добутку додатних елементів одновимірного масиву, та за її допомогою обробити п’ять масивів різної довжини.

13. Три групи студентів складали іспит з трьох предметів. Розробити функцію сортування списку студентів за спаданням оцінок за предметом і за її допомогою вивести відповідні повідомлення про студентів кожної групи.

14. Розробити функцію визначення найдовшого слова в рядку та з її допомогою отримати фразу із найдовших слів п’яти речень.

15. Розробити функції обчислення площ трикутника за формулою Герона та за формулою, що використовує основу трикутника та його висоту. Роботу функцій перевірити при визначенні площ декількох довільних трикутників.

16. Скласти функцію підрахунку кількості слів у реченні та використати її для обробки кількох речень.

17. Розробити функцію, що здійснює виділення в тексті слів паліндромів (тобто слів, які читаються однаково зліва направо та справа наліво), за її допомогою обробити введений текст.

18. Скласти функцію, що визначає в кожному стовпці матриці кількість від’ємних елементів та максимальний елемент. Використати цю функцію для обробки декількох матриць довільного розміру.

19. Розробити функцію, що визначає корені квадратного рівняння та перевіряє їх за теоремою Вієта, розв’язати за її допомогою декілька рівнянь.

20. Розробити функцію визначення частоти появи голосних літер у тексті та використати її для обробки декількох речень.

21. Розробити функцію визначення суми членів арифметичної прогресії та реалізувати її для трьох арифметичних прогресій з різною кількістю членів.

22. Розробити функцію визначення кількості появи заданої приголосної в тексті та застосувати цю функцію для обробки декількох речень.

23. Скласти функцію перестановки максимального елемента заданого стовпця матриці з мінімальним елементом заданого рядка матриці. Застосувати цю функцію для обробки трьох двовимірних масивів.

24. Розробити функцію, що здійснює сортування всіх елементів масиву за спаданням і знаходить добуток парних елементів, та за допомогою цієї функції обробити три одновимірних масиви довільної довжини.

25. Скласти функцію обчислення найбільшого спільного дільника двох чисел та продемонструвати її роботу для наборів пар чисел.

7. ФАЙЛИ

1. Створити файл і записати в нього список групи із зазначенням прізвища студента, статі і віку. Прочитати файл і вивести інформацію про студентів за заданими віком та статтю.

2. Створити файл, у який записати список студентів із зазначенням прізвища, ініціалів, назви групи, середнього балу, отриманого на сесії, і місця проживання. Вивести всі дані про студентів, середній бал яких не менше введеного.

3. Створити файл, у який записати бібліографічну інформацію про прочитані книги: автор, назва книги, рік видання. Прочитати цей файл і вивести назву книг, що видані у заданому користувачем році видання.

4. Створити файл, у який записати інформацію про студентів групи: прізвище, ініціали, рік народження, стать. Вивести інформацію про студентів, що народилися у зазначеному користувачем році.

5. Створити файл з повідомленнями про прізвища власників автомобілів, марки автомобілів, їхній колір та рік випуску. Вивести прізвища власників, що мають автомобіль зазначеного користувачем року випуску, марки та кольору.

6. Створити файл з такими повідомленнями про викладачів: прізвище, ініціали, назва кафедри, назва дисципліни, що викладає. Вивести повідомлення про викладачів з розташуванням їхніх прізвищ у алфавітному порядку.

7. Записати у файл матрицю **M[4][5]**, прочитати цей файл та вивести максимальні елементи другого рядка та четвертого стовпця матриці.

8. Створити файл, який містить номери потягів, їхній маршрут та час відправлення. При читанні цього файлу вивести повідомлення про маршрут проходження та час відправлення потяга із заданим користувачем номером.

9. Створити файл, який містить інформацію про студентів – харків'ян: прізвище, ініціали, середній бал за сесію, номер телефону та адресу. Вивести повідомлення про студентів за зростанням їхнього балу за сесію.

10. Створити файл, який містить повідомлення про студентів групи, що мають заборгованість по окремих предметах. Під час читання файлу вивести прізвища студентів за спаданням кількості заборгованостей.

11. Записати у файл довільну матрицю, прочитати отриманий файл та вивести матрицю, відсортовану за зростанням елементів рядків.

12. Записати у файл довільну квадратну матрицю, прочитати цей файл та вивести елементи матриці, розташовані вище побічної діагоналі.

13. Записати у файл таку інформацію: прізвище друга, дату та рік його народження та номер телефону. Вивести повідомлення про друзів за спаданням їхньої дати народження.

14. Записати у файл інформацію про друзів-однокурсників: прізвище, ім'я та номер телефону. Вивести повідомлення згідно з заданим номером телефону.

15. Записати в файл 20 дійсних чисел. Вивести число, записане під заданим порядковим номером та повідомлення про кількість таких чисел.

16. Створити файл, який містить список студентів і їх оцінки за результатами екзаменаційної сесії (не менше трьох предметів). Прочитати цей файл і вивести результати сесії заданого студента.

17. Створити файл, який містить повідомлення про студентів (прізвища, ініціали) та результати атестації з програмування. Вивести список неатестованих студентів.

18. Записати у файл список працівників із зазначенням прізвища, статі, року народження та спеціальності. Вивести список чоловіків-пенсіонерів.

19. Записати у файл довільну матрицю, вивести задані користувачем стовпці матриці, відсортовані за спаданням додатних елементів.

20. Записати у файл список спортсменів із зазначенням прізвища, статі та виду спорту. Вивести список чоловіків-волейболістів.

21. Створити файл з такою інформацією: прізвища студентів потоку, які проживають у гуртожитку, із зазначенням номерів кімнат та назви групи. Вивести інформацію про студентів, що проживають у кімнаті з заданим номером.

22. Створити файл, в який записати матрицю заданого розміру. Під час його читання підрахувати та вивести суму кожного рядка.

23. Записати у файл таку інформацію виробничого підрозділу: прізвище, стать та освіта працівника. Вивести прізвища жінок, які мають вищу освіту.

24. Записати у файл таку інформацію: прізвище та ініціали працівника підприємства, його посаду та зарплату. Вивести повідомлення про працівників, які мають зарплату меншу, ніж задана користувачем.

25. Записати у файл одновимірний масив, прочитати цей файл, вивести елементи, розташовані на парних місцях, та суму непарних елементів.

8. КЛАСИ

Для розв'язання задач усіх варіантів необхідно використовувати елементи **ООП**.

1. Обчислити площу рівностороннього трикутника та квадрата, якщо їхні сторони однакові.

2. Ввести список студентів групи, які проживають у гуртожитку. Вивести прізвища студентів, що мешкають у заданій кімнаті.

3. Обчислити площу круга та поверхню кулі, що мають однаковий радіус.

4. Ввести список студентів – харків'ян та вивести адреси студентів за заданими користувачем прізвищами.

5. Непарні елементи масиву **M[20]** записати в масив **M1**, а парні — в масив **M2**.

6. Ввести список студентів і їхню групу. Вивести список студентів заданої групи.

7. Обчислити площу квадрата й обсяг куба, що мають однакову довжину сторін.

8. Ввести відомість успішності студентів за трьома предметами. Вивести прізвища відмінників з математики та програмування.

9. У заданому масиві визначити максимальний і мінімальний елементи та їхні індекси.

10. Ввести відомість успішності студентів за трьома предметами. Вивести середній бал, отриманий усіма студентами з кожного предмета.

11. У матриці **M[5][5]** визначити добуток елементів, розташованих вище та нижче головної діагоналі.

12. Ввести відомість успішності студентів за трьома предметами. Вивести прізвища трьох студентів, у яких середній бал з усіх предметів найкращий.

13. Обчислити поверхню куба та площу прямокутника, в яких одна сторона однакова.

14. Ввести список студентів та дату їхнього народження (день, місяць, рік). Вивести прізвища студентів заданої дати народження.

15. У матриці **M[3][5]** обчислити суму елементів другого рядка та третього стовпця.

16. Ввести список студентів і їхню дату народження (день, місяць, рік). Вивести дату народження заданого студента.

17. У матриці **M[4][6]** визначити максимальний і мінімальний елементи та їхні індекси.

18. Ввести список студентів та дати їхнього народження (день, місяць, рік). Вивести прізвища студентів, які народилися у році, заданому користувачем.

19. Для циліндра з заданим радіусом і висотою обчислити площу поверхні та його об’єм.

20. Ввести список групи студентів (прізвища, стать, рік народження). Вивести прізвища студентів-юнаків заданого року народження.

21. У двох масивах однакової розмірності визначити максимальний елемент і вивести ім’я масиву, в якому цей елемент більший.

22. Ввести список студентів і оцінки за трьома предметами. Вивести цей список за спаданням їхнього середнього балу.

23. Обчислити площу рівностороннього трикутника та поверхню тристоронньої піраміди, побудованої з таких трикутників.

24. Ввести відомість успішності студентів за трьома предметами. Вивести успішність заданого студента з усіх предметів і середній бал.

25. Використовуючи інформацію про автомобілі (прізвище та ініціали власника, марку машини, її колір та номер), вивести прізвище власника автомобіля «Skoda» зеленого кольору.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Васильєв О.М. Програмування С++ в прикладах і задачах (2019).
2. Т.Г. Кормен, Ч.Е. Лейзерсон, Р.Л. Рівест, К. Стайн. Вступ до алгоритмів (2023).
3. Григорович В.Г. Алгоритмізація та програмування. Частина 1. (2023).
4. Григорович В.Г. Об'єктно-орієнтоване програмування. Частина 1. (2023).
5. Васильєв, О. Програмування на С++ в прикладах і задачах : навч. посіб. / Олексій Васильєв. – Київ : Ліра-К, 2020. – 381, [1] с. : іл.
6. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. Data Structures and Algorithms (1983).
7. Bjarne Stroustrup. Programming: Principles and Practice Using C++ (2009).
8. Donald E. Knuth. The Art of Computer Programming: Fundamental Algorithms (1997).
9. Exercises for Bjarne Stroustrup. The C++ Programming Language (4th Edition) (2013).
10. Harvey M. Deitel, Paul J. Deitel.C++ how to Program (10th Edition) (2017).
11. Jeremy A. Hansen.The Rook’s Guide to C++ (2013).
12. Kjell Bäckman. Structured Programming with C++ (2012).
13. Richard L. Halterman. Fundamentals of C++ Programming (2015).
14. Stanley Lippman, Josée Lajoie. C++ Primer (2012).
15. Stephen Prata. C++ Primer Plus, 6th Edition (Developer's Library) (2011).
16. Mastering C++ Programming Language: A Beginner’s Guide (2022).

Електронне навчальне видання

КОБИЛІН Олег Анатолійович
ЛЮБЧЕНКО Валентин Анатолійович
РУДЕНКО Діана Олександрівна
КИРИЧЕНКО Ірина Юріївна

Основи програмування мовою С++

Навчальний посібник
для студентів вищих навчальних закладів

Рецензенти:

М.О. Волк – д-р техн. наук, проф. кафедри електронних обчислювальних машин Харківського національного університету радіоелектроніки;

О.В. Золотухін – канд. техн. наук, доцент, зав. кафедри штучного інтелекту Харківського національного університету радіоелектроніки.

Редактор Б.П. Косіковська
Комп'ютерна верстка Л.Ю. Свєтайло
Відповідальний випусковий О.А. Кобилін

План 2025 (перше півріччя), поз. 2

Підп. до використання 04.02.2025 Формат pdf. Обсяг даних 1,4 Mb

ХНУРЕ. Україна. 61166, Харків, просп. Науки, 14, E-mail: info@nure.ua

Підготовлено в редакційно-видавничому відділі ХНУРЕ
Свідоцтво суб'єкта видавничої справи ДК №1409 від 26.06.2003