Міністерство освіти і науки України

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

Конспект лекцій (частина перша)
для здобувачів першого (бакалаврського) рівня вищої освіти
освітньої програми «Інформаційні системи
та технології охорони і безпеки»
галузь знань F Інформаційні технології
спеціальності F6 Інформаційні системи та технології
денної та заочної форм навчання

Луцьк 2025

УДК 004.421 (07)

А 45

Рекомендовано до видання вченою радою факультету КІТ ЛНТУ,
протокол № _____ від « ____» _____ 20___ року.

Голова вченої ради факультету КІТ _____ Інна КОНДІУС

Електронна копія друкованого видання передана для внесення в репозитарій ЛНТУ

Директор бібліотеки _____ Наталія ПОЛІЩУК

Розглянуто і схвалено на засіданні кафедри комп'ютерної інженерії та безпеки
ЛНТУ, протокол № _____ від « ____» _____ 20____ року.

Завідувач кафедри КІБ _____ Тарас ТЕРЛЕЦЬКИЙ

Укладач: _____ Світлана ЛАВРЕНЧУК, кандидат технічних наук, доцент кафедри комп'ютерної інженерії та безпеки ЛНТУ

Рецензент: _____ Петро ПЕХ, кандидат технічних наук,
доцент кафедри комп'ютерної інженерії та безпеки ЛНТУ

Відповідальний за випуск: _____ Тарас ТЕРЛЕЦЬКИЙ, кандидат технічних наук, доцент кафедри комп'ютерної інженерії та безпеки ЛНТУ

А 45 **Алгоритмізація та програмування**: конспект лекцій (частина перша) для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Інформаційні системи та технології охорони і безпеки» галузь знань F Інформаційні технології спеціальності F6 Інформаційні системи та технології денної та заочної форм навчання / уклад. С. В. Лавренчук. Луцьк: ЛНТУ, 2025. 192 с.

Конспект лекцій (частина перша) з дисципліни «Алгоритмізація та програмування» складений відповідно до діючої програми курсу.

Призначений для здобувачів вищої освіти спеціальності Інформаційні системи та технології освітньої програми «Інформаційні системи та технології охорони і безпеки».

ЗМІСТ

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням обсягів даних, складністю інформаційних систем та підвищеними вимогами до їхньої надійності, ефективності й безпеки. У цих умовах особливого значення набувають фундаментальні знання з алгоритмізації та програмування, що формують базу для подальшої професійної підготовки фахівців у галузі інформаційних систем та технологій охорони і безпеки.

Метою вивчення дисципліни *«Алгоритмізація та програмування»* є формування у здобувачів освіти системних знань про принципи побудови алгоритмів, основні елементи структурного програмування та практичні навички реалізації алгоритмів.

Перша частина конспекту лекцій присвячена мовам програмування **C** і **C++**. Здобувачі повинні опанувати синтаксис і семантику базових конструкцій мови, набути умінь роботи з масивами, функціями, вказівниками, структурами та файлами, що є необхідною основою для розробки ефективних програмних рішень у сфері інформаційних технологій.

Перша частина курсу структурована за двома змістовими модулями:

Змістовий модуль 1. *Алгоритми та основи мови C/C++* охоплює теоретичні засади побудови програм, типи даних, оператори, функції, алгоритми розгалуження та повторення, роботу з масивами.

Змістовий модуль 2. *Алгоритми роботи з вказівниками та абстрактними типами даних* присвячений питанням організації пам’яті, роботі з вказівниками, текстовими даними, структурами, об’єднаннями та файлами.

Вивчення навчального матеріалу сприяє формуванню у студентів компетентностей, необхідних для аналізу, розроблення та супроводу програмних компонентів інформаційних систем, а також забезпечує підґрунтя для подальшого освоєння сучасних мов і технологій програмування.

Матеріали конспекту лекцій містять систематизований теоретичний виклад основних тем курсу, приклади програмного коду та контрольні питання.

ТЕМА 1. Загальні відомості про мови програмування. Структура програми мовою С

1.1 Синтаксис комп'ютерної мови

Можна сказати, що кожна мова (машинна чи природна, не має значення) складається з таких елементів:

– алфавіт – набір символів, які використовуються для побудови слів певної мови (наприклад, Латинський алфавіт для Англійської, Кирилиця для Української, Кандзі для Японської тощо);

– лексика (словник) – набір слів, які мова пропонує своїм користувачам (наприклад, слово «computer» походить зі словника англійської мови, а слово «cmoptrue» – ні; слово «chat» присутнє як в англійських, так і в французьких словниках, але значення у них різні);

– синтаксис – набір правил (формальних чи неформальних, записаних чи інтуїтивно зрозумілих), які використовуються для визначення того, чи утворює певна послідовність слів коректне речення (наприклад, «I am a student» є синтаксично вірною фразою, тоді як «I a student am» є синтаксично невірною);

– семантика – набір правил, які визначають, чи має певна фраза смисл (наприклад, фраза «Я з'їв пончик» має смисл, а фраза «Пончик з'їв мене» немає).

Формальними називають мови, які вигадані людьми для вирішення специфічних завдань.

Наприклад, набір спеціальних знаків і правил запису формул, що використовуються математиками для запису формул і доказів теорем, є формальною мовою. Хіміки так само використовують свою формальну мову для запису хімічної структури речовин. *Мови програмування – формальні мови, призначені для опису алгоритмів.*

Формальні мови характерні тим, що мають чіткі синтаксичні правила.

Наприклад, `3+5=8` є синтаксично правильним математичним записом, а `3=+5$` – ні. H_2O – синтаксично правильна хімічна формула речовини, а `3Ff` – ні.

Коли ви читаєте речення українською мовою або вираз на формальній мові, ви визначаєте його структуру, часто не усвідомлюючи. Цей процес як ми знаємо з уроків української мови називається синтаксичним аналізом або синтаксичним розбором.

Наприклад, коли ви читаєте фразу «*ІТішники вчать програмування*», ми за пропусками визначаємо початок і кінець слів і лише після цього знаходимо підмет («ІТішники») і присудок («вчать»). Розібравши синтаксичну структуру, ви можете зрозуміти її зміст – семантику.

Будь-який транслятор перед тим, як перетворити програму в зрозумілий для комп'ютера вигляд, виконує синтаксичний аналіз. При синтаксичному аналізі транслятор розбирає синтаксичну структуру виразів, і знаходить так звані символи (*tokens*) – синтаксично неподільні частки. У даному контексті символами можуть бути назви змінних, числа, знаки операцій, ключові слова, позначення хімічних елементів.

Використання символу *$* у формулі *3=+5$* не має сенсу, і це є однію з причин, чому вона невірна з точки зору математики. Та ж проблема у формулі з хімії 3*Ff*: у таблиці Менделєєва немає елементу з позначенням *Ff*. Другий тип синтаксичних помилок пов'язаний з неправильною структурою виразів, тобто послідовністю дотримання символів. Вираз *3=+5$* має невірну структуру, оскільки відразу після знаку рівності не може слідувати знак додавання. Аналогічно, в молекулярних формулах використовуються нижні індекси, але вони не можуть йти перед позначенням хімічного елементу.

Хоча формальні і природні мови мають багато спільного, вони мають ряд важливих відмінностей:

1. Однозначність

У природних мовах безліч ідіом і метафор; часто люди визначають значення фраз залежно від ситуації, в якій вони використовуються. Формальні мови розроблені так, щоб виключити неоднозначність виразів. Це означає, що *вираз повинен мати лише одне значення незалежно від контексту*.

2. Надмірність

Для того, щоб позбавитися від неоднозначності і уникнути нерозумінь в природних мовах використовуються доповнення. *Формальні мови короткі і максимально виразні.*

3. Важливість структури і безпомилковість

У формальних мовах дуже важлива структура, тому читання справа наліво або знизу догори – не кращий спосіб їх зрозуміти. Маленькі орфографічні і пунктуаційні помилки (і друкарські помилки також), які в природних мовах можуть бути проігноровані, у формальних мовах матимуть велике значення, аж до зміни змісту на протилежний.

З цих відмінностей можна виділити основні *властивості, якими має володіти алгоритм, програма.*

Мова програмування – це штучна мова створена для того щоб взаємодіяти, «спілкуватися» з комп'ютером.

Мови програмування діляться на 2 типи: високорівневі і низькорівневі. Високорівневі мови – це мови які зрозумілі для сприйняття людиною, в свою чергу низькорівневі мови програмування близькі до машинного коду. Також мови мають різні парадигми такі як: функціональна, об'єктно-орієнтована, процедурна або аспектно-орієнтована і т.д.

Синтаксис комп'ютерної мови – це сукупність правил, що визначають комбінації символів, які вважаються правильно структурованим документом або фрагментом цієї мови. Це стосується як мов програмування, результатом якого є початковий код, так і мов розмітки, результатом якого є дані.

Синтаксис мови визначає її *форму*. Текстові комп'ютерні мови базуються на послідовностях символів, а візуальні мови програмування – на просторовому макеті та зв'язках між символами (які можуть бути текстовими чи графічними).

Синтаксис комп'ютерної мови, як правило, розрізняють на три рівні:

– слова – лексичний рівень, що визначає, як символи утворюють лексеми;

– фрази – рівень граматики, який визначає, як лексеми утворюють фрази;

– контекст – визначення, на що посилаються імена об’єктів або змінних, якщо типи є придатними, тощо.

Такий підхід забезпечує модульність, яка дозволяє описувати та обробляти кожен рівень окремо і часто незалежно. По-перше, лексер перетворює лінійну послідовність символів у лінійну послідовність лексем (також відомо як «лексичний аналіз»). По-друге, аналізатор перетворює лінійну послідовність лексем у ієрархічне дерево синтаксису (відомо як «синтаксичний аналіз»). По-третє, контекстуальний аналіз визначає імена та перевіряє типи.

Синтаксис визначає граматично правильні слова, семантика наділяє програми змістом, а прагматика визначає призначення і умови застосування мови програмування.

Написана мовою програмування високого рівня програма називається *вихідним кодом* (на відміну від машинного коду, що виконується комп’ютерами). Так само файл, що містить вихідний код, називається *вихідним файлом*.

1.2 Компілятори та інтерпретатори штучних мов

Комп’ютерне програмування – це компонування елементів обраної мови програмування у порядку, який призведе до бажаного результату. Ефект може бути різним у кожному конкретному випадку – це залежить від фантазії, знань і досвіду програміста.

Звичайно, таке компонування має відповідати багатьом критеріям:

– алфавіту – програма має бути написана розпізнаваним шрифтом, наприклад латиницею, кирилицею тощо.

– лексично – кожна мова програмування має свій словник і його потрібно опанувати; на щастя, він набагато простіший і менший за словник будь-якої природної мови;

– синтаксично – у кожній мові є свої правила, і їх потрібно дотримуватись;

– семантично – програма має мати певний смисл.

На жаль, програміст також може робити помилки за будь-яким з чотирьох вищевказаних критеріїв. Кожна з них може призвести до того, що програма стане абсолютно не придатною до використання або не працюватиме зовсім.

Дослідники, що вивчають питання появи розуму на нашій планеті, вважають, що вирішальну роль у його розвитку відіграла поява мови, яка дозволила не тільки виражати і зберігати знання, але й обмінюватися ними. Зі створенням комп'ютерів виникла потреба в спілкуванні з подібними пристроями, оскільки виявилося необхідним передавати їм накази, завдання й описи роботи, які вони повинні виконувати. Для цієї мети почали розробляти спеціальні мови, які стали називатися штучними на відміну від природних мов спілкування людей. Штучні мови повинні бути, з одного боку, зручними і зрозумілими для людини, а з іншого – повинні сприйматися пристроями. Сполучення цих вимог в одній мові виявилося важкою задачею, тому з'явилися засоби для перетворення текстів з мови, зрозумілої людині, на мову пристрою. Такі засоби назвали *трансляторами*.

Транслятор може бути інтерпретуючого чи компілюючого типу. У першому випадку його називають *інтерпретатором* вхідної мови, а в другому – *компілятором*. Інтерпретатор послідовно читає пропозиції вхідної мови, аналізує їх і відразу ж виконує, а компілятор не виконує пропозиції мови, а будує програму, що може надалі бути запущена для одержання результату. На вхід компілятора подається текст, написаний вхідною мовою – мовою, зрозумілою людині, а результатом роботи компілятора є текст мовою, зрозумілою пристрою.

Робота компілятора (транслятора) складається з декількох стадій, що можуть виконуватися послідовно, або сполучатися за часом.

Перша стадія роботи компілятора називається *лексичним аналізом*, а програма реалізації – лексичним аналізатором. На вхід лексичного аналізатора подається послідовність символів вхідної мови. Лексичний аналізатор виділяє в цій послідовності найпростіші конструкції мови, які називають *лексичними*

одиницями. Прикладами лексичних одиниць є *ідентифікатори*, *числа, символи операцій, службові слова* і т.д. Лексичний аналізатор перетворює вихідний текст, заміняючи лексичні одиниці їх внутрішнім поданням – лексемами. *Лексема* може включати інформацію про клас лексичної одиниці і її значення. Крім того, для деяких класів лексичних одиниць лексичний аналізатор будує таблиці, наприклад, таблицю ідентифікаторів, констант, що використовуються на наступних стадіях компіляції.

Другу стадію роботи компілятора називають *синтаксичним аналізом*, а відповідну програму – синтаксичним аналізатором. На вхід синтаксичного аналізатора подається послідовність лексем, що потім перетвориться в проміжний код. Кожен атом проміжного коду включає опис операції, яку потрібно виконати, та містить вказівку на операнди, що використовуються. При цьому послідовність розташування атомів, на відміну від лексем, відповідає порядку виконання операцій, необхідному для одержання результату.

Третю стадію роботи називають *семантичним аналізом*. У процесі семантичного аналізу перевіряється, наскільки коректне сумісне розташування компонентів програми, накопичується інформація про типи даних і здійснюється перевірка типів операндів.

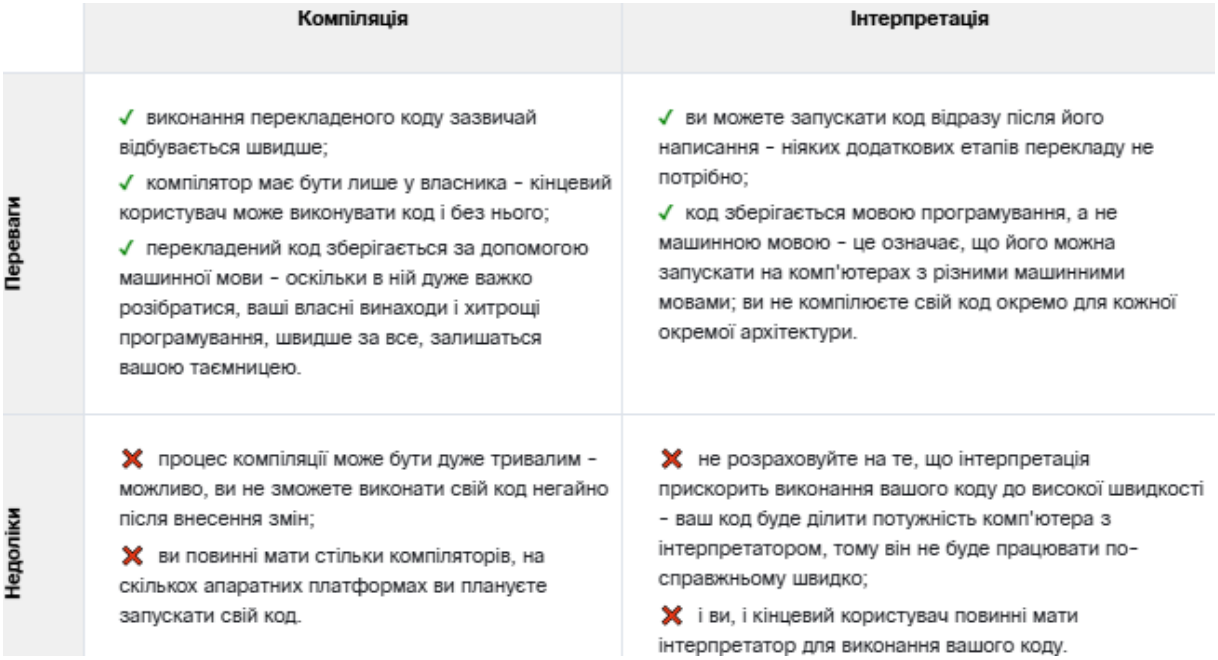
На четвертій стадії роботи компілятора генерується проміжний код, а на п’ятій стадії проходить його оптимізація. На шостій стадії здійснюється побудова вихідного тексту. Програма, що реалізує цю стадію, називається генератором вихідного тексту. Генератор кожному символу дії, що надходить на його вхід, ставить у відповідність одну чи кілька команд вихідної мови. Як вихідна мова можуть бути використані команди пристрою, команди асемблера або оператори якої-небудь іншої мови.

Порівняння інтерпретаторів та компіляторів штучних мов з точки зору використання програмістами наведено на рисунку 1.1.

Отож, мови програмування бувають компільовані і інтерпретовані.

Якщо програма написана компілюємою мовою (C, C++, Pascal), перед виконанням її потрібно повністю перевірити наявність синтаксичних помилок і

вже після цього перевести в зрозумілу для комп’ютера форму – машинний код. Це робить спеціальна програма, яка називається компілятором.



	Компіляція	Інтерпретація
Переваги	✓ виконання перекладеного коду зазвичай відбувається швидше; ✓ компілятор має бути лише у власника - кінцевий користувач може виконувати код і без нього; ✓ перекладений код зберігається за допомогою машинної мови - оскільки в ній дуже важко розібратися, ваші власні винаходи і хитрощі програмування, швидше за все, залишаться вашою таємницею.	✓ ви можете запускати код відразу після його написання - ніяких додаткових етапів перекладу не потрібно; ✓ код зберігається мовою програмування, а не машинною мовою - це означає, що його можна запускати на комп'ютерах з різними машинними мовами; ви не компілюєте свій код окремо для кожної окремої архітектури.
Недоліки	✗ процес компіляції може бути дуже тривалим - можливо, ви не зможете виконати свій код негайно після внесення змін; ✗ ви повинні мати стільки компіляторів, на скількох апаратних платформах ви плануєте запускати свій код.	✗ не розраховуйте на те, що інтерпретація прискорить виконання вашого коду до високої швидкості - ваш код буде ділити потужність комп'ютера з інтерпретатором, тому він не буде працювати по-справжньому швидко; ✗ і ви, і кінцевий користувач повинні мати інтерпретатор для виконання вашого коду.

Рисунок 1.1 – Переваги та недоліки компіляції та інтерпретації

Якщо програма написана мовою, що інтерпретується (Python, PHP, Ruby), вона не перекладається в машинний код повністю. Натомість спеціальна програма, яка називається інтерпретатором, йде за кодом, аналізує його та виконує кожну окрему команду.

Існують мови програмування, які поєднують обидва підходи (**C#, Java**). У таких мовах код вихідної програми спочатку компілюється в проміжний код (байт-код), а вже потім, під час виконання, переводиться у машинний код.

1.3 Етапи розв’язування задач

Етапи розв’язування задач включають кілька послідовних кроків (рисунок 1.2), що охоплюють як постановку задачі, так і її програмну реалізацію та перевірку.

1. Під час аналізу умови задачі здійснюємо:

– визначення мети задачі;

– формулювання вхідних та вихідних даних;

– з’ясування обмежень та особливостей (діапазон значень, точність обчислень, час виконання тощо).

Рисунок 1.2 – Етапи розв’язування задач

2. Алгоритмізація передбачає:

– розробку послідовності дій (алгоритму), що забезпечує розв’язання задачі;

– використання блок-схем, псевдокоду або словесного опису алгоритму;

– перевірку правильності алгоритму на тестових прикладах «вручну».

3. Під час структуризації програми здійснюємо:

– вибір методів реалізації: які структури даних та функції використовувати;

– поділ програми на модулі, функції, класи (у разі потреби);

– визначення змінних, їхніх типів, а також допоміжних структур (масиви, структури, вектори тощо).

4. Під час кодування (програмування):

– запис алгоритму (мовою C/C++ або іншою);

– використання конструкцій мови: операторів, виразів, умовних розгалужень, циклів, функцій, класів;

– забезпечення читабельності коду (іменування, коментарі, форматування).

5. На етапі компіляції та налагодження відбувається:

– перевірка синтаксису та виправлення помилок компіляції;

– використання налагоджувачів (debugger) для пошуку логічних помилок;

– тестування програми на простих і граничних вхідних даних.

6. Тестування передбачає:

– перевірку правильності роботи програми на різних наборах даних;

– аналіз результатів, пошук і виправлення можливих недоліків алгоритму чи реалізації.

7. Документування та оформлення включає:

– написання коментарів у коді;

– складання інструкцій щодо використання програми;

– за потреби – підготовку технічної документації.

8. Аналіз ефективності містить етапи:

– оцінки часу виконання, використання пам’яті;

– оптимізації алгоритмів і коду (за необхідності).

Ці 8 етапів під час навчання можна об’єднати три основні великі групи:

1. Постановка задачі і розробка алгоритму.
2. Складання програми.
3. Виконання програми, тестування і аналіз результатів.

1.4 Структура програми мовою С.

Суттєвою особливістю мови С++ порівняно з іншими мовами є те, що програми складаються з функцій, які відіграють роль підпрограм в інших мовах. Головна функція, яка має бути у кожній програмі, – це функція вигляду:

```
main(void)
{
тіло функції з командою return 0;
}
```

де `main()` – заголовок функції. Ключове слово `void` означає, що функція не залежить від параметрів, його записувати не обов'язково. Функцію з параметрами розглядатимемо нижче. У тілі функції містяться команди та виклики інших функцій. Команди одну від одної відокремлюють символом `";"` (крапка з комою). Текст функції закінчується командою повернення `return`. Тіло функції (усі команди після заголовка) записується у фігурних дужках `{ }`.

Розглянемо програму, у результаті виконання якої на екран буде виведено повідомлення: Привіт студентам групи ІСТО від С++! (лістинг 1.1).

Лістинг 1.1 – Моя перша програма мовою С++

```
// Моя перша програма мовою С++
#include <iostream.h>
int main()
{
cout <<" Привіт студентам групи ІСТО від С++!";
return 0;
    }
```

Кінець лістингу 1.1

Директива `#include <iostream.h>` приєднує бібліотечний файл `iostream.h`. Саме у цьому файлі описані функції, які дають змогу виконувати операції введення-виведення даних.

Далі у програмі записана обов'язкова функція `main()`. Ключове слово `int`

означає, що функція `main()` повертатиме у точку виклику результат цілого типу.

Конструкція `cout <<` забезпечує виведення на екран монітора повідомлення «Привіт студентам групи ІСТО від С++!».

Команда `return` слугує для виходу з функції `main().` Числовий параметр після `return` є результатом (значенням) функції (у цій програмі – 0).

Зверніть увагу. Функцію `main()` можна застосувати так:

```
void main()
{
тіло функції;
}
```

Така функція називається функцією `main()` типу `void`. Вона не повертає у програму жодних значень, тому команду `return` писати не треба.

Загальна структура програми.

Найпростіша програма мовою С++ має такий загальний вигляд, як наведено в лістингу 1.2.

Лістинг 1.2 – Найпростіша програма мовою С++

```
// коментарі
#include <назва бібліотечного файла>
void main()
{
<тіло функції>;
    }
```

Кінець лістингу 1.2

Ви вже знаєте, що в кутових дужках < > записують значення параметра директиви, які у програмі пропускати не можна. Крім цього, у кутових дужках описуватимемо словами загальні конструкції мови, замість яких у реальній програмі будуть конкретні команди. У цьому разі у програмі кутові дужки не пишуть.

Зазвичай складніші програми містять значну кількість функцій та

додаткових елементів. Тому у загальному випадку програма мовою С++ має такий вигляд, як в лістингу 1.3.

Лістинг 1.3 – Загальна структура програми мовою С++

```
// коментарі
#include <назва бібліотечного файлу 1>
...
#include <назва бібліотечного файлу N>
<інші директиви препроцесора>
<оголошення глобальних змінних>;
<оголошення глобальних сталих>;
<оголошення та створення функцій користувача>;
<тип результату функції> main(опис формальних параметрів)
{
<оголошення локальних змінних>
<оголошення локальних сталих>;
<команди>;
    }
```

Кінець лістингу 1.3

Розрізняють глобальні та локальні дані. Дані, визначені для всіх функцій, тут називатимемо глобальними, а дані, які використовуються лише в окремих функціях чи блоках, – локальними.

1.5 Алфавіт мови С++

Мову програмування С++ на початку 80-х років створив Бьяртні Страуструп на базі популярної серед професіоналів-програмістів мови С, яку розробив Деніс Рітчі. Мова одержала назву від С та операції інкременту (++), визначеної лише у ній. Така операція збільшує на одиницю значення змінної, до якої її застосовують.

С++ є розширенням мови С. Окрім стандартних команд, сюди увійшли засоби для об’єктно-орієнтованого й узагальненого програмування. С++ – це перша у світі мова об’єктно-орієнтованого програмування, суть якого полягає в об’єднанні даних та алгоритмів їх опрацювання у єдине ціле.

Компілятор мови С++ сприймає початковий файл, що містить програму

мовою С++ як послідовність текстових рядків. Компілятор мови С++ послідовно прочитує рядки програми і розбиває кожен з прочитаних рядків на групи символів, які називаються лексемами. Лексема – одиниця тексту програми, яка має самостійний зміст для компілятора мови і не містить інших лексем. Потім компілятор на основі граматики мови розпізнає смислові конструкції мови (вирази, визначення, описи, оператори і т. д.), побудовані з цих лексем. В С++ є шість типів лексем: ідентифікатори, ключові слова, константи, рядкові літерали, знаки операції та роздільники.

Алфавіт мови включає літери, цифри, спеціальні знаки.

Букви включають малі і великі букви латинського алфавіту, причому компілятор розглядає одну й ту саму велику і рядкову букви як різні символи.

Як цифри використовуються арабські цифри: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Букви та цифри використовуються під час формування констант, ідентифікаторів і ключових слів.

Спеціальні знаки перелічені в таблиці 1.1.

Таблиця 1.1 – Спеціальні знаки

Символ	***Найменування***	***Символ***	***Найменування***
,	Кома	!	Знак оклику
.	Крапка	|	Вертикальна риска
;	Крапка з комою	/	Похила риска праворуч (слеш)
:	Двокрапка	\	Похила риска ліворуч (зворотний слеш)
?	Знак питання	~	Тильда
′	Поодинока лапка (апостроф)	_	Підкреслення
(	Ліва кругла дужка	#	Знак номера
)	Права кругла дужка	%	Відсоток
{	Ліва фігурна дужка	&	Амперсанд
}	Права фігурна дужка	^	Стрілка вгору
<	Знак «менше»	–	Знак мінус
>	Знак «більше»	=	Знак рівності
[	Ліва квадратна дужка	+	Знак плюс
]	Права квадратна дужка	*	Знак множення (зірочка)
″	Лапки		

Компілятор, виконуючи лексичний аналіз тексту програми мовою С++, для розпізнавання початку і (або) кінця окремих лексем використовує пробільні символи.

Пробільні символи – пробіл, символи табуляції, переведення рядка,

переведення каретки, вертикальної табуляції. Ці символи відділяють лексеми, наприклад, константи та ідентифікатори. До пробільних символів належать коментарі.

Крім перерахованих знаків у мові С використовуються керуючі послідовності (Esc-послідовності). Керуючі послідовності – спеціальні символи, що дозволяють представляти пробільні та неграфічні символи у символьних і рядкових константах. Їх застосовують для представлення символів або чисел, які не можна безпосередньо ввести з клавіатури. Для введення керуючих послідовностей, що дозволяють отримати візуальне представлення деяких символів, що не мають графічного аналога, використовується символ зворотної похилої риски (\). У таблиці 1.2. перелічені допустимі керуючі послідовності мови С++.

Таблиця 1.2 – Керуючі послідовності мови С++

Код	**Найменування**	**Десяткове**	**Шістнадцят-кове**	**Позначуваний символ**
\a	Звуковий сигнал	7	0x07	BEL
\b	Повернення на один крок	8	0x08	BS
\f	Переведення на наступну сторінку	12	0x0C	FF
\n	Перехід на наступний рядок	10	0x0A	LF
\r	Переведення каретки	13	0x0D	CR
\t	Горизонтальна табуляція	9	0x09	HT
\v	Вертикальна табуляція	11	0x0B	VT
\\	Зворотна коса риса	92	0x5C	\
\'	Апостроф	39	0x27	′
\"	Подвійні лапки	34	0x22	″
\?	Знак питання	63	0x3F	?
\000	Вісімковий код символу	Будь-який	Будь-який	Будь-який
\x00	Шістнадцятковий код символу	Будь-який	Будь-який	Будь-який

Ідентифікатори – це умовні позначення змінних, функцій тапозначок, інших об’єктів, що визначаються користувачем, використовуваних у програмі. Ідентифікатор може складатися з послідовності однієї чи кількох літер, цифр та знаків підкреслювання. Вони можуть бути практично будь-якими завдовжки.

У мові **С** (С89/С90) стандарт гарантує розрізнення *мінімум* 31 символа для внутрішніх і 6 – для зовнішніх ідентифікаторів. У **С++** (починаючи з першого стандарту) немає обмежень: ідентифікатори розрізняються повністю.

Ідентифікатор може включати прописні та рядкові букви, але вони не повинні збігатися за написанням з ключовими словами. Наприклад, `Name, name, NAME` – це абсолютно різні ідентифікатори.

Ідентифікатори змінних не повинні:

1) починатися з цифри;

2) містити пробіли;

3) містити спеціальні символи за винятком символу підкреслення _ ;

4) збігатися з ключовими словами мови C++ і з іменами бібліотечних функцій.

Назви змінних не повинні починатися з цифри, містити пробіли, спеціальні символи та ключові слова в C++, збігатися з назвами бібліотек функцій.

Нижче наведено кілька прикладів коректних і некоректних імен ідентифікаторів.

Коректні	***Некоректні***
`Value`	`3value`
`test5`	`re!salt`
`h_balance`	`h-balance`
`_word`	`$word`

Ключові слова – це ідентифікатори, які мають певний зміст для компілятора мови C++. Вони зарезервовані в мові для спеціального використання.Ключові слова завжди пишуться малими буквами. У таблиці 1.3 наведені ключові слова мови C++.

Таблиця 1.3 – Основні ключові слова мови C++

`asm`	`double`	`mutable`	`switch`
`auto`	`else`	`new`	`template`
`bool`	`enum`	`operator`	`this`
`break`	`explicit`	`private`	`throw`
`case`	`extern`	`protected`	`try`
`catch`	`float`	`public`	`typedef`

`char`
`class`
`const`
`continue`
`default`
`delete`
`do`
	`for`
	`friend`
	`goto`
	`if`
	`inline`
	`int`
	`long`
		`register`
		`return`
		`short`
		`signed`
		`sizeof`
		`static`
		`struct`
			`Typename`
			`Union`
			`unsigned`
			`virtual`
			`void`
			`volatile`
			`while`

1.6 Коментарі в мові С++

Коментарі – послідовність символів, яка ігнорується компілятором і сприймається ним як окремий пробільний символ.

Коментарі – повідомлення, які пояснюють нам, що робить програма в тому чи іншому місці. Коментарі потрібно використати, коли необхідно пояснити яку-небудь операцію коду. Коментарі завжди мають нести певне смислове навантаження, а не перефразовувати операції мови. Писати коментарі слід паралельно з програмою, що створюється. Існує два способи включення коментарів до програми: традиційний метод С і метод С++. Традиційний коментар С має такий вигляд:

```
/*<послідовність символів >*/
```

Коментарі можуть займати більше за один рядок, але не можуть бути вкладеними. Наприклад:

```
/* Це однорядковий коментар в стилі С */
/ * Це багаторядковий коментар
у стилі С */
```

У С++ коментар починається з подвійної похилої риски (//) і продовжується до кінця поточного рядка. Наприклад:

```
// Коментарі допомагають документувати програму.
```

Часто програмісти використовують стиль С для багаторядкових коментарів, а для коротких зауважень використовують однорядкові коментарі стилю С++.

1.7 Сталі та змінні, типи даних

Змінна чи стала – це пойменована ділянка оперативної пам’яті комп’ютера, де зберігається значення деякої величини.

Змінні і сталі (їх прийнято називати даними) мають такі властивості: назву (ім’я), значення, тип. Назву дає програміст.

Для роботи з даними слід зарезервувати певний обсяг оперативної пам'яті комп'ютера, де зберігатимуться їхні значення. Тому всі дані, які використовуються у програмі, потрібно заздалегідь описати (оголосити), оскільки компілятор розподіляє пам'ять згідно з описами.

Якщо значення деякої величини (даного) не змінюватиметься протягом виконання усієї програми, то таке дане варто задати як *сталу* (константу, `const`). Це можна зробити так:

```
const <назва сталої 1 > = значення сталої 1 >;
```

або так:

```
const <тип> <назва сталої 2> = значення сталої 2>;
```

Сталу 2 називають типізованою сталою. За замовчуванням числова стала належить до цілого типу. Під час виконання програми значення сталих змінювати не можна.

Приклад 1. Оголосимо три сталі

```
const vik = 20, rist = 176;
const float g = 2.78;
```

Для сталої `g` задано тип – `float` і значення `2,78`.

У C++ є такі стандартні сталі: число π є сталою `M_PI`, `π/2- M_PI_2`, `π/4- M_PI_4`, `1/π - M_1_PI`, `-M_1_SQRTPI` тощо. Ці сталі можна безпосередньо використовувати у програмі, заздалегідь підключивши модуль `math.h`.

Дані, які під час виконання програми можуть набувати різних значень, називаються *змінними*. їх оголошують так:

```
<тип змінних 1> <список змінних1>;
<тип змінних N> <список змінних N>;
```

Елементи списків записують через кому. Наприклад, змінні оголошують так:

```
int a, c; float b, d, z; char w;
```

Змінним можна задавати початкові значення відразу під час оголошення. Це називається ініціалізацією даних. Наприклад,

```
float b, d = 2.5, a = 4;
char w = ‘t’;
```

Отже, у загальному випадку змінні одного типу оголошують так:

```
<тип змінних> <назва змінної 1> = значення 1>,
...
<назва змінної N> = значення N>,
<список інших змінних>;
```

1.8 Стандартні функції введення/виведення

Для перевірки працездатності та правильності роботи програми потрібно певним чином відобразити результати її виконання, а також організувати введення необхідних для виконання програми даних.

Мова C/C++ не містить вбудованих (таких, що входять до складу компілятора) засобів, які б виконували введення/виведення даних. Всі операції обміну даними як з термінальними пристроями (дисплей, клавіатура тощо), так і з дисковими файлами, найчастіше реалізують через звертання до відповідних бібліотечних функцій.

Програмування процесів введення та виведення даних засобами мов C та C++ істотньо відрізняються. Тому ми спочатку розглянемо це питання стосовно мови C, а далі – стосовно мови C++.

1.8.1 Форматне виведення даних засобами мови C

У мові **C** форматне виведення даних здійснюється за допомогою стандартної функції

```
printf("формат", список_аргументів);
```

Функція `printf()` визначена в заголовковому файлі `<stdio.h>` і дозволяє виводити текст разом із відформатованими значеннями змінних.

Перший аргумент функції `printf()` – *рядок формату,* який може містити звичайний текст та *специфікатори формату*.

Кожен специфікатор формату починається зі знака % і визначає, як саме потрібно відобразити відповідне значення (таблиця 1.4).

Таблиця 1.4 – Основні специфікатори формату

Специфікатор	***Тип даних***	***Приклад виводу***
%d, %i	Ціле число (int)	123
%u	Беззнакове ціле	255
%ld	Довге ціле (long int)	123456
%f	Число з плаваючою точкою	3.141593
%0.2f	Число з 2 знаками після коми	3.14
%e, %E	Експоненційна форма	1.23e+02
%c	Символ (char)	A
%s	Рядок (char*)	Hello
%p	Адреса (вказівник)	0x7ffee3b2c4
%%	Символ %	%

Приклади використання наведено в лістингу 1.4.

Лістинг 1.4 – Форматований вивід даних мовою С

```
#include <stdio.h>

int main() {
    int a = 42;
    float pi = 3.14159;
    char c = ‘X’;
    char str[] = "C programming";
    printf("Ціле число: %d\n", a);
    printf("Число з плаваючою точкою: %.2f\n", pi);
    printf("Символ: %c\n", c);
    printf("Рядок: %s\n", str);

    return 0;
    }
```

Кінець лістингу 1.4

Вивід:

```
Ціле число: 42
Число з плаваючою точкою: 3.14
```

```
Символ: X
Рядок: C programming
```

Ще приклади форматного виводу наведено в таблиці 1.5.

Таблиця 1.5 – Приклади використання специфікаторів форматування

Запис	Пояснення	Приклад	Результат
`%d`	вивід цілого числа	`printf("%d", 42);`	42
`%5d`	мінімальна ширина поля 5 (вирівнювання вправо)	`printf("%5d", 42);`	42
`%-5d`	мінімальна ширина поля 5 (вирівнювання вліво)	`printf("%-5d", 42);`	42
`%05d`	доповнення числа нулями зліва	`printf("%05d", 42);`	00042
`%f`	число з плаваючою точкою (6 знаків після крапки за замовчуванням)	`printf("%f", 3.14);`	3.140000
`%.2f`	2 знаки після крапки (точність)	`printf("%.2f", 3.14);`	3.14
`%10.2f`	ширина 10, точність 2	`printf("%10.2f", 3.14);`	3.14
`%-10.2f`	ширина 10, точність 2, вирівнювання вліво	`printf("%-10.2f", 3.14);`	3.14
`%s`	вивід рядка	`printf("%s", "Hello");`	Hello
`%10s`	рядок у полі шириною 10 (вирівнювання вправо)	`printf("%10s", "Hi");`	Hi
`%-10s`	рядок у полі шириною 10 (вирівнювання вліво)	`printf("%-10s", "Hi");`	Hi

1.8.2 Форматне введення даних у стилі C

Форматне введення даних у стилі C передбачає використання функції `scanf()` та її похідних для зчитування даних із стандартного вводу (клавіатури) за певним форматом.

Синтаксис `scanf()`:

```
int scanf(const char *format, ...);
```

`format` – рядок формату, який визначає, який тип даних і як очікується

зчитати. Інші параметри – адреси змінних, куди будуть записані введені значення. Приклади наведено в таблиці 1.6.

Таблиця 1.6 – Основні специфікатори формату та введення даних

Специфікатор	**Тип даних**	**Приклад**
%d	int	scanf("%d", &a);
%i	int (десяткова, вісімкова, шістнадцяткова)	scanf("%i", &a);
%f	float	scanf("%f", &x);
%lf	double	scanf("%lf", &y);
%c	char	scanf(" %c", &ch); *(зверни увагу на пробіл перед %c)*
%s	char[]	scanf("%s", str); *(записує до пробілу)*
%u	unsigned int	scanf("%u", &n);
%x / %X	int (шістнадцяткова)	scanf("%x", &num);
%o	int (вісімкова)	scanf("%o", &num);

Примітка: перед змінною обов'язково ставиться &(адреса), крім масивів. Розглянемо приклади форматного введення (лістинги 1.5-1.7).

Лістинг 1.5 – Введення двох чисел мовою С

```
#include <stdio.h>
int main() {
    int a, b;
    printf("Введіть два числа: ");
    scanf("%d %d", &a, &b);
    printf("Ви ввели: %d і %d\n", a, b);
    return 0;
    }
```

Кінець лістингу 1.5

Лістинг 1.6 – Введення рядка мовою С

```
#include <stdio.h>
int main() {
    char name[50];
    printf("Введіть ім'я: ");
    scanf("%49s", name); // обмеження до 49 символів
    printf("Привіт, %s!\n", name);
    return 0;
    }
```

Кінець лістингу 1.6

Лістинг 1.7 – Введення символу з пропуском пробілів мовою С

```
#include <stdio.h>
int main() {
    char ch;
    printf("Введіть символ: ");
    scanf(" %c", &ch); /* пробіл перед %c ігнорує попередні пробіли/нові рядки*/
    printf("Ви ввели: %c\n", ch);
    return 0;
    }
```

Кінець лістингу 1.7

Особливості форматного введення:

– `scanf()` зупиняється на пробілі, табуляції або символі нового рядка (для `%s`);

– для безпечного зчитування рядків використовуйте обмеження довжини (`%49s`), щоб уникнути переповнення буфера;

– перед `%c` ставлять пробіл, щоб пропустити залишки попереднього вводу;

– `scanf()` повертає кількість успішно зчитаних елементів, що можна перевіряти на помилки.

1.8.3 Введення-виведення в стилі C++

У C++ введення та виведення здійснюється в стилі C++ через *потоки* (`iostream`) замість функцій `scanf()/printf()` стилю C. Це більш безпечний і зручний спосіб роботи з даними. Базові об’єкти потоків наведено в таблиці 1.7.

Таблиця 1.7 – Базові об’єкти потоків та їх призначення

Об’єкт потоку	***Призначення***
`cin`	введення з клавіатури
`cout`	виведення на екран
`cerr`	виведення помилок на екран (не буферизується)
`clog`	логування (буферизоване виведення помилок)

Для їх використання необхідно підключити заголовок `iostream` (лістинг 1.8):

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

Синтаксис введення даних:

```
cin >> змінна1 >> змінна2 >> ...;
```

Лістинг 1.8 – Введення даних мовою С++

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

int main() {
    int a;
    double b;
    cout << "Введіть ціле число та дійсне число: ";
    cin >> a >> b;
    cout << "Ви ввели: " << a << " і " << b << endl;
    return 0;
    }
```

Кінець лістингу 1.8

Особливості:

– >> автоматично пропускає пробіли, табуляції та символи нового рядка;

– підходить для чисел, символів і рядків (але для рядків без пробілів — cin >> str).

4. Синтаксис виведення даних:

```
cout << дані1 << " " << дані2 << endl;
```

Приклад наведено в лістингу 1.9.

Лістинг 1.9 – Введення даних мовою С++

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

int main() {
    string name;
    cout << "Введіть ім’я: ";
    cin >> name;
```

```
    cout << "Привіт, " << name << "!" << endl;
    return 0;
    }
```

Кінець лістингу 1.9

Особливості:

– << дозволяє «склеювати» дані для виведення;

– endl – символ нового рядка та скидання буфера;

– можна використовувати маніпулятори з <iomanip> для форматування (setw, setprecision, fixed).

Приклад з форматуванням наведено в лістингу 1.10.

Лістинг 1.10 – Виведення числа з точністю 2 знаки після коми

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

int main() {
    double pi = 3.14159265;
    cout << fixed << setprecision(2);
    cout << "Pi = " << pi << endl; // Pi = 3.14
    return 0;
    }
```

Кінець лістингу 1.10

Вирівнювання тексту:

```
cout << setw(10) << "Привіт" << endl;
```

Контрольні питання

1. Що таке мова програмування і для чого вона використовується?
2. Які існують покоління мов програмування? Наведіть приклади.
3. У чому полягає різниця між компіляторами та інтерпретаторами?
4. Які основні етапи розв’язування задачі за допомогою комп’ютера?
5. Що таке алгоритм і які властивості він має?
6. Яка загальна структура програми мовою С?
7. Яке призначення функції main() у програмі мовою С?
8. Що входить до алфавіту мови С++?
9. Яке призначення коментарів у програмі? Які види коментарів підтримує мова С++?

10. Що таке змінна? Як відбувається її оголошення та ініціалізація?

11. Які типи даних підтримуються в мові С (основні, похідні, користувацькі)?

12. Чим відрізняються сталі (константи) від змінних? Як оголошуються сталі в мові С++?

13. Яке призначення операторів введення та виведення? Які стандартні функції використовуються (scanf, printf, cin, cout)?

14. Що таке оператор, вираз і інструкція в мові програмування?

15. Як здійснюється компіляція та виконання програми мовою С (етапи трансляції та відладки)?

ТЕМА 2. Арифметичні операції, оператори мови С

2.1 Типи даних, перетворення типів

Кожне дане програми (змінна чи константа) має певний тип. Тип задає *обсяг оперативної пам'яті*, яка виділяється для збереження цього даного, визначає *діапазон допустимих значень* даного та встановлює *операції*, які можуть виконуватись з цим даним.

Тип змінних у С/С++ -програмах завжди вказується явно в їх описах, а тип констант встановлюється компілятором автоматично за формою запису константи в програмі.

Всі типи мови С/С++ можна поділити на три групи:

– скалярні (або прості) типи, які в довільний момент часу можуть мати тільки одне значення; скалярними є арифметичні типи, переліки та вказівники;

– агреговані (або складені) типи, які формуються за встановленими правилами з наборів скалярних типів; до агрегованих типів належать: масиви, структури та об'єднання;

– тип «функція», що оголошує функції із заданим складом параметрів і встановленим типом значення, яке повертає функція.

Арифметичні типи можна поділити ще на дві групи:

– цілочислові типи, дані яких можуть набувати тільки цілих значень (знакових або беззнакових) і зберігаються в форматах, встановлених для цілих чисел;

– дійсні типи, дані яких є дійсними числами і зберігаються в оперативній пам'яті в форматах чисел з плаваючою крапкою.

Крім цього, типи даних поділяють на базові (інші назви: основні, фундаментльні) та похідні.

Базовими вважаються типи: `char, int, float, double`, а також типи, утворені з них через застосування модифікаторів: `short, long, signed, unsigned`.

Особливим базовим типом є `void` – порожній або невизначений тип. Тип

`void`, зокрема, застосовують в оголошенні функцій, щоб зазначити, що функція не повертає ніякого значення, або щоб вказати; що функція не використовує параметрів.

Похідними вважаються вказівникові типи, агреговані типи та функції. Всі вони формуються з використанням базових типів.

1. Прості (базові) типи:
   1) цілі числа:
      - `short` (коротке ціле);
      - `int` (ціле число);
      - `long` (довге ціле);
      - модифікатори: `signed, unsigned;`
   2) дійсні числа (з плаваючою крапкою):
      - `float` (одинарна точність);
      - `double` (подвійна точність);
      - `long double` (розширена точність);
   3) символьний тип
      - `char` (зберігає один символ або його ASCII-код).
2. Складені типи
   - масиви;
   - структури (`struct`);
   - об'єднання (`union`);
   - перелічувані типи (`enum`);
3. Спеціальні типи
   - `void` (відсутність значення/типу);
   - вказівники (адреси пам'яті).

Згідно зі стандартом мови C/C++ розмір типу `int` відповідає розміру машинного слова, властивому програмно-апаратній платформі конкретної системи програмування. Для переважної більшості сучасних комп'ютерів розмір становить 2 або 4 байти. До типу int можна застосувати модифікатори long (довге ціле) та `short` (коротке ціле). Модифікатор типу `long` вказує, що

дане буде займати не менше 4 байтів. Модифікатор `short` вказує, що дане не повинно перевищувати розмір типу `int` (у більшості реалізацій тип short int займає 2 байти). Більш детально типи даних мови С та їх розміри наведено в таблиці 2.1.

Таблиця 2.1 – Типи даних у С

Тип даних	**Розмір (байт)**	**Діапазон значень (приблизно)**
`char`	1	від –128 до 127 (signed); від 0 до 255 (unsigned)
`short`	2	від –32 768 до 32 767
`unsigned short`	2	від 0 до 65 535
`int`	4	від –2 147 483 648 до 2 147 483 647
`unsigned int`	4	від 0 до 4 294 967 295
`long`	4 (на 32-біт)8 (на 64-біт)	–2 147 483 648 … 2 147 483 647 (32-біт)–9.22e18 … 9.22e18 (64-біт)
`unsigned long`	4 / 8	0 … 4.29e9 (32-біт)0 … 1.84e19 (64-біт)
`long long`	8	–9.22e18 … 9.22e18
`unsigned long long`	8	0 … 1.84e19
`float`	4	~1.2e–38 … 3.4e38 (точність ~6–7 знаків)
`double`	8	~2.3e–308 … 1.7e308 (точність ~15 знаків)
`long double`	10 / 16	~3.4e–4932 … 1.1e4932 (точність ~18–19 знаків, залежно від компілятора)

Конкретні значення діапазонів залежать від компілятора (наприклад, GCC, MSVC) і архітектури системи. Для дійсних чисел (`float, double`) вказано не діапазон точних значень, а приблизні межі.

Неявне перетворення (*implicit conversion*) типів відбувається автоматично при виконанні виразів:

– менший тип → більший тип (ціле → дійсне, `int` → `float`).

– `char` автоматично перетворюється в `int` під час обчислень.

Приклад:

```
int a = 5;
float b = 2.5;
```

```
float c = a + b;//a автоматично перетворюється у float
```

Проте, коли ми напишемо:

```
cout<<5/2;
```

отримаємо результат 2 (а не 2,5 як очікували). Тут компілятор приводить до цілого типу результат, оскільки тип даних явно не вказаний.

Якщо ми хочемо отримати дробове число, то можна написати так:

```
cout<<5./2;
```

або

```
cout<<5/2.0;
```

або

```
cout<<float(5)/2;
```

В останньому прикладі використано явне перетворення (e*xplicit conversion, casting*). Воно здійснюється програмістом за допомогою операторів приведення типів.

Синтаксис:

```
(новий_тип) вираз
```

Приклади:

```
int a = 5, b = 2;
float c;

// неявне: a/b = 2 (цілочисельне ділення)
c = a / b;
printf("%.2f\n", c);  // 2.00

// явне: перетворюємо a в float
c = (float)a / b;
printf("%.2f\n", c);  // 2.50
```

При змішаних операціях типи перетворюються «у більший» за такою ієрархією: `char → int → long → float → double → long double`.

2.2 Поняття оператора. Типи операторів

У мовах програмування, зокрема C, *оператор* – це елементарна (окрема) інструкція для комп'ютера, яка визначає певну дію, що має бути виконана під час роботи програми.

Оператор – найменша виконувана одиниця програми.

Кожен оператор у C зазвичай закінчується *крапкою з комою* (`;`).

Послідовність операторів виконується у порядку їх розташування, якщо не задано інше (цикли, умовні переходи тощо).

Класифікація операторів у мові C:

1) Прості оператори:
   - оператор виразу (наприклад: `a = b + c;`)
   - порожній оператор (`;`)
2) Складені оператори (блоки)
   - об'єднання кількох операторів у фігурні дужки `{ … }`.

Складені оператори використовуються для групування дій, наприклад у тілі функції, циклу, умовного переходу.

3) Оператори управління послідовністю виконання
   - умовні: `if, if...else, switch`;
   - циклічні: `for, while, do...while`;
   - переривання: `break, continue, goto, return`.

Приклад наведено в лістингу 2.1.

Лістинг 2.1 – Приклад використання операторів розгалуження

```
#include <stdio.h>

int main() {
    int a = 5, b = 3, c;

    c = a + b;        // оператор присвоєння
    if (c > 5) {      // умовний оператор
        printf("c більше за 5\n");    /* оператор виклику функції*/
```

```
    }

    return 0;          // оператор виходу з функції
    }
```

Кінець лістингу 2.1

У цьому прикладі кожен рядок з `;` або конструкцією керування є окремим оператором.

2.3 Арифметичні операції з числами

В мові **C** арифметичні операції (таблиця 2.2) дозволяють виконувати дії над числовими значеннями (**цілими** та **дійсними**).

Таблиця 2.2 – Основні арифметичні операції

Оператор	**Приклад**	**Дія**
`+`	`a + b`	додавання
`-`	`a - b`	віднімання
`*`	`a * b`	множення
`/`	`a / b`	ділення (для цілих – ціла частина від ділення)
`%`	`a % b`	остача від ділення (лише для цілих чисел)

Особливості:

Додавання та віднімання (+, -) виконуються як для цілих (**`int`**), так і для дійсних (**`float, double`**) чисел.

Множення (*) працює для будь-яких числових типів.

Ділення (/):

– якщо обидва операнди цілі – результат буде цілим (дробова частина відкидається):

```
7 / 2 = 3;
```

– якщо хоча б один операнд дійсний – виконується ділення з плаваючою точкою:

```
7.0 / 2 = 3.5;
```

Остача від ділення (%):

– використовується лише для цілих чисел;

– результат має той самий знак, що і ділене;

```
7 % 3 = 1
-7 % 3 = -1
```

Приклад наведено в лістингу 2.2.

Лістинг 2.2 – Приклад використання арифметичних операторів

```
#include <stdio.h>

int main() {
    int a = 7, b = 2;
    float x = 7.0, y = 2.0;

    printf("a + b = %d\n", a + b);   // 9
    printf("a - b = %d\n", a - b);   // 5
    printf("a * b = %d\n", a * b);   // 14
    printf("a / b = %d\n", a / b);     /* 3 (цілочисельне
ділення)*/
    printf("a %% b = %d\n", a % b);  // 1

    printf ("x / y = %.2f\n", x / y);   /* 3.50 (цілочисельне
ділення)*/


    return 0;
    }
```

Кінець лістингу 2.2

Таким чином, арифметичні операції в С поділяються на *базові* (+ - * /) і спеціальні (%), з урахуванням різниці між роботою з цілими та дійсними числами.

2.4 Операції присвоєння. Повна та скорочена форма операції присвоєння

Повна форма операції присвоєння – це базова операція, яка записує значення виразу в змінну.

Синтаксис:

змінна = вираз;

– ліва частина (змінна) – це *l-value* (об’єкт, що має адресу в пам’яті);

– права частина (вираз) – це *r-value* (значення, яке обчислюється);

– тип значення виразу має бути сумісним з типом змінної.

Приклад:

```
int a, b, c;
a = 5;        // присвоєння константи
b = 3;        // присвоєння константи
c = a + b;    // присвоєння результату виразу
```

Скорочена форма (комбіновані операції присвоєння) – це запис операції, яка одночасно виконує дію (додавання, віднімання, множення тощо) і присвоює результат змінній.

Загальна форма:

```
змінна op= вираз;
```

де `op` – арифметичний чи побітовий оператор.

Приклади:

– `a += b;` → еквівалентно `a = a + b;`

– `a -= b;` → еквівалентно `a = a - b;`

– `a *= b;` → еквівалентно `a = a * b;`

– `a /= b;` → еквівалентно `a = a / b;`

– `a %= b;` → еквівалентно `a = a % b;`

– `a <<= 2;` → еквівалентно `a = a << 2;` (зсув вліво)

– `a >>= 2;` → еквівалентно `a = a >> 2;` (зсув вправо)

– `a &= b;` → еквівалентно `a = a & b;` (побітове І)

– `a |= b;` → еквівалентно `a = a | b;` (побітове АБО)

– `a ^= b;` → еквівалентно `a = a ^ b;` (побітове виключне АБО)

Приклад коду наведено в лістингу 2.3.

Лістинг 2.3 – Приклад використання комбінованих операцій присвоєння

```
#include <stdio.h>

int main() {
    int a = 10, b = 3;

    a = a + b;   // повна форма
    printf("a = %d\n", a);  // 13

    a += b;      // скорочена форма (a = a + b)
    printf("a = %d\n", a);  // 16

    a *= 2;      // скорочена форма (a = a * 2)
    printf("a = %d\n", a);  // 32

    return 0;
    }
```

Кінець лістингу 2.3

Отже, *повна форма* використовується для загальних випадків, а *скорочена форма* зручна для компактного запису та підвищує читабельність коду.

Можна записати вираз, який буде складатись із декількох присвоєнь:

```
u = w = z = beg
```

Асоціативність операцій присвоєння (справа наліво) визначає порядок виконаня присвоєнь. У результаті всі три змінні `z`, `w` і `u` почергово набудуть значення змінної `beg`, це значення буде також загальним значенням усього виразу.

2.5 Операції інкремента та декремента

У C/C++ програмах широко застосовують унарні операції присвоєння `++` та `--`.

Перша з них, яка називається *інкрементом*, збільшує значення свого операнда на 1.

Друга операція називається *декрементом*, вона зменшує на 1 значення

заданого операнда.

Мова C/C++ використовує дві форми операцій інкремента та декремента: префіксну, коли знак ++ / -- записується перед операндом, і постфіксну, коли знак операції вказується після операнда.

Якщо унарне присвоєння використовується як окремий оператор, то обидві форми: префіксна та постфіксна рівнозначні. Всі чотири наступних оператори виконують ту ж саму дію: збільшують на 1 значення змінної k:

```
++k;          k++;          k += 1;          k =k+1;
```

Відповідно зменшити на 1 значення к можна кожним з наступних операторів:

```
- -k;          k- -;          k -= 1;          k=k-1;
```

Різниця між префіксною і постфіксною формами операцій інкремента та декремента проявляється, коли вони використовуються як операнди інших операцій. Префіксний інкремент/декремент встановлює, що спочатку значення змінної збільшується чи, відповідно, зменшується на 1, а вже потім вона вступає в іншу операцію. Якщо ж використовується постфіксна форма, то змінна спочатку віддає у вираз своє значення, а вже потім збільшується/зменшується на 1.

Приклад наведено в лістингу 2.4.

Лістинг 2.4 – Приклад використання операцій інкременту та декременту

```
#include <stdio.h>

int main() {
    int a = 5, b;

    b = ++a;   // префіксний інкремент
    printf("a = %d, b = %d\n", a, b); // a=6, b=6

    a = 5;     // скинемо значення
    b = a++;   // постфіксний інкремент
    printf("a = %d, b = %d\n", a, b); // a=6, b=5
```

```
    a = 5;
    b = --a;   // префіксний декремент
    printf("a = %d, b = %d\n", a, b); // a=4, b=4

    a = 5;
    b = a--;   // постфіксний декремент
    printf("a = %d, b = %d\n", a, b); // a=4, b=5

    return 0;
    }
```

Кінець лістингу 2.4

2.6 Стандартні математичні функції

Стандартні математичні функції у C++ описані в бібліотеці `math.h`, тому її необхідно приєднати на початку програми:

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

Аргументи функцій записують у круглих дужках. Основні математичні функції наведені в таблиці 2.3.

Таблиця 2.3 – Математичні функції мови C++

Назва функції	**Математичний запис**	**Назва функції**	**Математичний запис**
`abs(x)`	$\|x\|$ (для цілих чисел)	`pow(x,y)`	x^y
`fabs(x)`	$\|x\|$ (для дійсних чисел)	`pow10(x)`	10^x
`cos(x)`	$\cos(x)$	`sqrt(x)`	$\sqrt{x}$
`sin(x)`	$\sin(x)$	`exp(x)`	e^x
`tan(x)`	$tg(x)$	`ceil(x)`	заокруглює число x до більшого цілого
`acos(x)`	$ar\cos(x)$		
`asin(x)`	$\arcsin(x)$	`floor(x)`	відкидає дробову частину числа x
`atan(x)`	$arctg(x)$		
`log(x)`	$\ln(x)$	`fmod(x,y)`	обчислює остачу від ділення числа x на число y
`log10(x)`	$\lg(x)$		

Примітка. З курсу математики відомо, що $\sqrt[n]{x} = x^{\frac{1}{n}}$, а мовою C++ такий вираз можна записати використовуючи функцію `pow(x,y)`.

Контрольні питання

1. Що таке оператор у мові C?
2. Який оператор використовується для присвоєння значення змінній?
3. Який результат виконання виразу `x = 5 + 3 * 2;`?
4. У якому порядку виконуються арифметичні операції в мові C?
5. Який оператор означає збільшення значення змінної на 1?
6. Як записати скорочену форму операції `a = a + 5;`?
7. Який тип результату має операція ділення двох цілих чисел `int` у мові C?
8. Що відбудеться при виконанні `5 / 2` у мові C?
9. Який оператор використовується для піднесення до степеня?
10. Яка бібліотека містить математичні функції (наприклад `sqrt, pow, sin`)?
11. Який результат операції `7 % 3`?

ТЕМА 3. Алгоритми розгалуження, умовні оператори

3.1 Поняття алгоритму

Під алгоритмом розуміють сукупність правил функціонування, що описують поведінку деякої системи, керуючись якими вона досягає кінцевого результату.

Алгоритм має скінченну множину кроків, кожен з яких може виконати одну або більше операцій. Операції мають бути однозначними та ефективними. Кожен крок слід виконувати за скінченний, у межах розумного, час. Алгоритм повинен мати хоча б один вихід і нуль або більше зовнішніх входів та закінчуватись після скінченного числа операцій.

Алгоритм, алгорифм (латинізов. Algorithmi, від імені узб. математика IX ст. аль-Хорезмі) – система правил виконання обчислювального процесу, що обов'язково приводить до розв'язання певного класу задач після скінченного числа операцій. При написанні комп'ютерних програм алгоритм описує логічну послідовність операцій. Для візуального зображення алгоритмів часто використовують блок-схеми.

Поняття алгоритму належить до первісних понять математики, таких, як поняття множини чи натурального числа. Обчислювальні процеси алгоритмічного характеру (арифметичні дії над цілими числами, знаходження найбільшого спільного дільника двох чисел і т. п.) відомі людству з глибокої давнини. Проте в явному вигляді поняття алгоритму сформувалося лише на початку 20 століття.

Під алгоритмом звичайно розуміють скінченну множину точно визначених правил для чисто механічного розв'язку задач певного класу. Алгоритми мають наступні характерні властивості:

– *скінченність* – алгоритм є скінченним об'єктом, що є необхідною умовою його механічної реалізовності;

– *масовість* – початкові дані для алгоритму можна вибирати із певної (можливо, нескінченної) множини даних; це означає, що алгоритм призначений не для однієї конкретної задачі, а для класу однотипних задач;

– *дискретність* – розчленованість процесу виконання алгоритму на окремі кроки; це означає, що алгоритмічний процес здійснюється в дискретному часі;

– *елементарність* – кожен крок алгоритму має бути простим, елементарним, можливість виконання якого людиною або машиною не викликає сумнівів;

– *детермінованість* – однозначність процесу виконання алгоритму; це означає, що при заданих початкових даних кожне дане, отримане на певному (не початковому) кроці, однозначно визначається даними, отриманими на попередніх кроках;

– *результативність, визначеність* – алгоритм має засоби, які дозволяють відбирати із даних, отриманих на певному кроці виконання, результативні дані, після чого алгоритм зупиниться.

Алгоритм – поетапний процес, щоб отримати рішення для чітко визначеної проблеми.

Для кожної проблеми або класу завдань може бути багато різних алгоритмів. Для кожного алгоритму може бути багато різних реалізацій (програм) (рисунок 3.1).

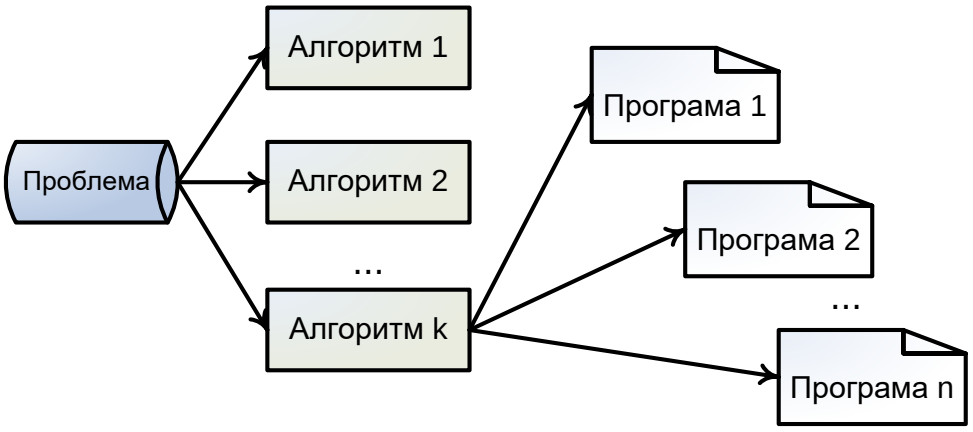


Рисунок 3.1 – Проблеми, алгоритми та програми

3.2 Способи представлення алгоритмів

Способи опису алгоритмів:

– формульний;

– словесний;

– графічний;

– табличний;

– алгоритмічний.

Алгоритми можуть бути виражені в багатьох типах позначень, таких як природні мови, псевдокод, блок-схеми та мови програмування.

Вирази природних мов алгоритмів, як правило, бувають багатослівними та неоднозначними, і вони рідко використовуються для складних або технічних алгоритмів.

Псевдокоди та блок-схеми – це структуровані способи вираження алгоритмів, які уникають багатьох неоднозначностей, що є загальними в природних мовних виразах, уникаючи деталей виконання.

Мови програмування, в основному, призначені для вираження алгоритмів у формі, яку може виконувати комп’ютер, але вони часто використовуються як спосіб визначення алгоритмів або їх документування.

Псевдокод по суті є англійською, з деякими визначеними правилами структури та деякими ключовими словами, які змушують його виглядати як код програми. Деякі рекомендації щодо написання псевдокоду виглядають таким чином.

Ключові слова, які використовуються для псевдокоду:

– для початку та закінчення `BEGIN MAINPROGRAM, END MAINPROGRAM`;

– для ініціалізації `INITIALISATION, END INITIALISATION`;

– для підпрограми `BEGIN SUBPROGRAM, END SUBPROGRAM`;

– для вибору `IF, THEN, ELSE, ENDIF`;

– для вибору багатоканального режиму `CASEWHERE, OTHERWISE, ENDCASE`;

– для циклу з передумовою `WHILE, ENDWHILE`;

– для циклу з післяумовою `REPEAT, UNTIL`;

Ключові слова записуються великими літерами.

Структурні елементи складаються парами, наприклад, для кожного `BEGIN` є `END`, для кожного `IF` є `ENDIF` і т.д.

Відступ використовується для показу структури алгоритму.

Назви підпрограм підкреслюються. Це означає, що при уточненні рішення задачі слово в алгоритмі може бути підкреслено та розроблена підпрограма. Ця функція полягає в тому, щоб допомогти використати концепцію розвитку «зверху».

Блок-схеми – це схематичний метод представлення алгоритмів. Вони використовують інтуїтивно зрозумілу схему показу операцій у вікнах, пов'язаних лініями та стрілками, які графічно показують потік керування в алгоритмі. Стандарти для блок-схеми показують, що основний напрямок потоку приймається як зверху вниз і зліва направо.

Програми обчислювальних досліджень вказують на структуру програмування послідовності, вибору, повторення та підпрограм. Описується кожна з цих структур разом з прикладами їх використання.

Кожна з п'яти прийнятних структур може бути побудована з основних елементів, як показано нижче (рисунки 3.2–3.4).

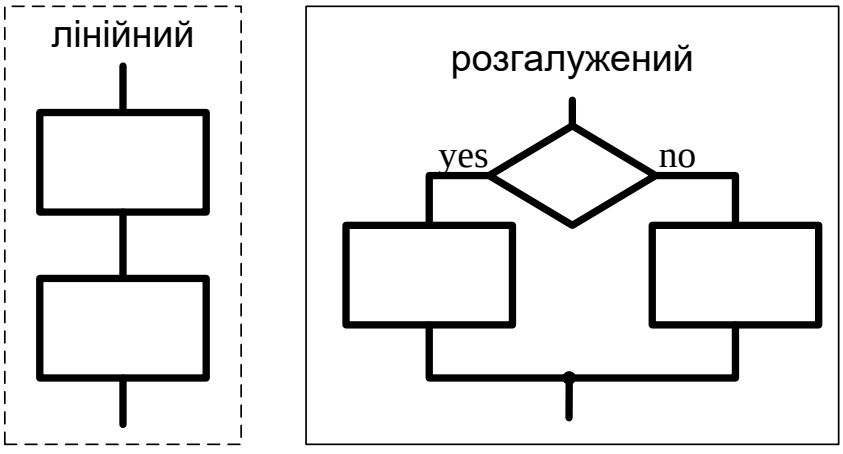


Рисунок 3.2 – Лінійний та розгалужений

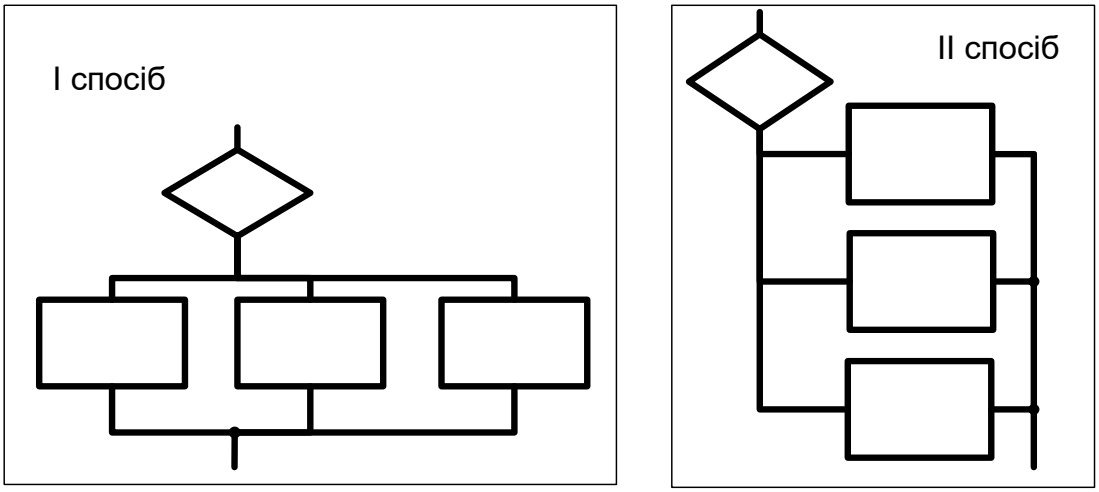


Рисунок 3.3 – Багатопозиційний вибір

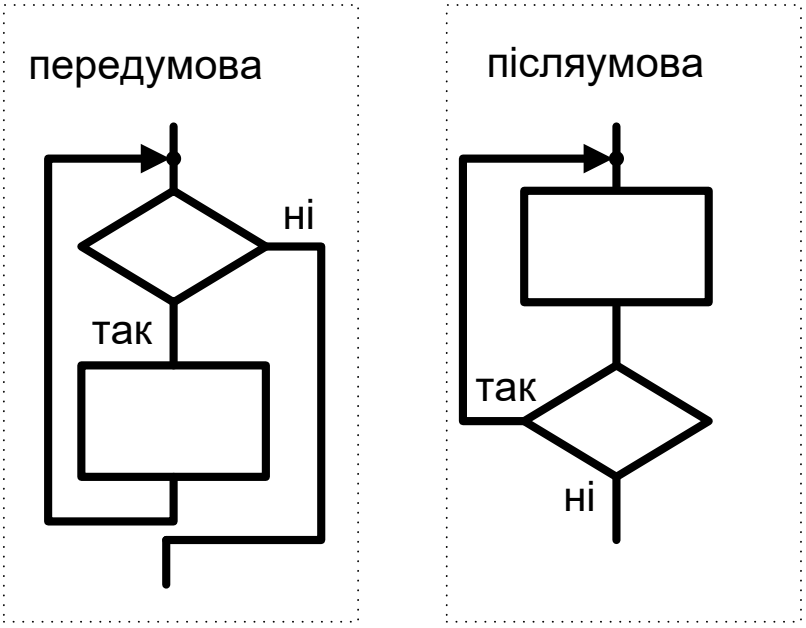


Рисунок 3.4 – Цикли

У всіх випадках відзначимо, що є лише одна точка входу до структури та одна точка виходу.

У комп’ютерній програмі або алгоритмі послідовність включає в себе прості кроки, які повинні виконуватися один за іншим. Ці кроки виконуються в тому ж порядку, в якому вони написані.

Спробуємо вивчити алгоритм описання, використовуючи приклади.

Лінійний алгоритм

Завдання 1. Розробити алгоритм для додавання двох чисел та відображення результату (блок-схема до псевдокоду наведена на рисунку 3.5).

```
BEGIN
get values of a & b
c ← a + b
display c
END
```

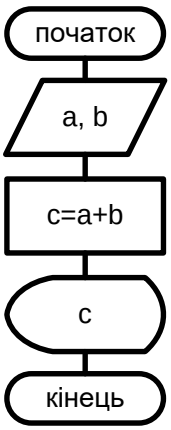


Рисунок 3.5 – Блок-схема лінійного алгоритму

Розгалужені алгоритми

Завдання 2. Створіть алгоритм реагування на телефон (блок-схема до псевдокоду наведена на рисунку 3.6).

```
IF the telephone is ringing THEN
   answer the telephone
ENDIF
```

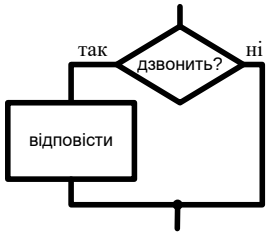


Рисунок 3.6 – Блок-схема розгалуженого алгоритму

Завдання 3. Створіть алгоритм реагування на світлофор (червоне або зелене світло) (блок-схема до псевдокоду наведена на рисунку 3.7).

```
IF the signal is green THEN
    proceed through the intersection
ELSE
    stop the vehicle
ENDIF
```

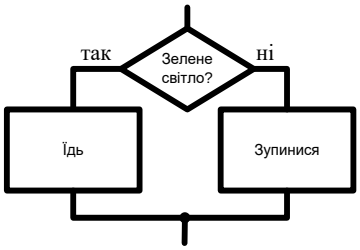


Рисунок 3.7 – Блок-схема повного розгалуженого алгоритму

Використання багатопозиційного вибору

Завдання 4. Створіть алгоритм реагування на світлофор (врахувати всі варіанти) (блок-схема до псевдокоду наведена на рисунку 3.8).

```
CASEWHERE signal is
   red : stop
   yellow : get ready
   green : go through the intersection
   OTHERWISE : go with caution
ENDCASE
```

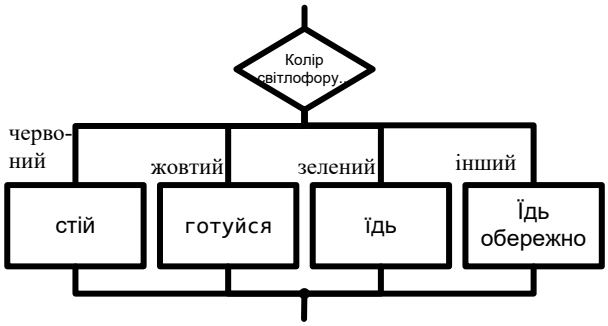


Рисунок 3.8 – Блок-схема алгоритму багатопозиційного вибору

3.3 Розгалуження

У мовах програмування розгалуження (branching) використовується для організації *вибору шляху виконання програми* залежно від умови.

3.3.1 Оператор if

Цей оператор виконує блок коду, якщо умова істинна (ненульове значення).

```
if (умова) {
    // оператори, якщо умова істинна
}
```

Приклад:

```
int a = 5;
if (a > 0) {
    printf("a додатне\n");
}
```

3.3.2 Оператор if ... else

Цей оператор дозволяє вибрати між двома варіантами.

```
if (умова) {
    // якщо умова істинна
} else {
    // якщо умова хибна
}
```

Приклад:

```
int a = -3;
if (a >= 0) {
    printf("a невід’ємне\n");
} else {
    printf("a від’ємне\n");
}
```

3.3.3 Вкладені if (багатоступеневий вибір)

Використовується для перевірки кількох умов.

```
if (a > 0) {
    printf("Додатне\n");
} else if (a < 0) {
    printf("Від’ємне\n");
} else {
    printf("Нуль\n");
}
```

3.3.4 Оператор switch

В мові **C** оператор вибору – це `switch ... case`. Він застосовується, коли потрібно вибрати одну з кількох альтернатив залежно від значення виразу (як правило, цілого або символьного типу).

```
switch (вираз) {
    case значення1:
        // дії, якщо вираз == значення1
        break;
    case значення2:
        // дії, якщо вираз == значення2
        break;
    ...

    default:
        // дії, якщо жодне значення не підійшло
}
```

Приклад:

```
int day = 3;
switch (day) {
    case 1: printf("Понеділок\n"); break;
    case 2: printf("Вівторок\n"); break;
    case 3: printf("Середа\n"); break;
    default: printf("Невідомий день\n");
}
```

`switch` працює швидше та зручніше за багато вкладених `if-else`, якщо є багато варіантів.

Після кожного `case` зазвичай ставиться `break;`, щоб уникнути «провалювання» у наступний блок.

`default` виконується, якщо жоден `case` не збігся (аналог `else`).

3.4 Тернарний оператор (?)

У мові *C* тернарний оператор `?` – це скорочена форма умовного розгалуження, він виконує вибір між двома значеннями.

```
вираз = (умова) ? значення1 : значення2;
```

Якщо **умова істинна** → береться значення1; якщо **умова хибна** → береться значення2.

Приклад:

```
int a = 7;
int b = (a % 2 == 0) ? 0 : 1;  // b = 1, бо a непарне
```

Тернарний оператор `?` – компактний запис `if...else`, зручний для простих виборів між двома значеннями.

Давайте порівняємо два приклади визначення, чи число парне (таблиця 3.1).

Таблиця 3.1 – Приклад застосування `if...else` та тернарного оператора

Варіант з `if-else`	Оператор `?`

`int a = 7;`
`int b;`
 	
`if (a % 2 == 0) {`
`    b = 0;`
`} else {`
`    b = 1;`
`}`
	`int a = 7;`
	`int b;`
	
	`b = (a % 2 == 0) ? 0 : 1;`

Отже, варіант `if-else` підходить для більш складних дій, коли потрібно виконати кілька операторів, а оператор `?` використовується для *коротких виборів* між двома значеннями в одному рядку.

3.5 Вкладені оператори розгалуження

Можна використовувати *вкладені оператори розгалуження* – це коли один `if, switch` чи інший умовний оператор знаходиться всередині іншого (рисунок 3.9, лістинг 3.1).

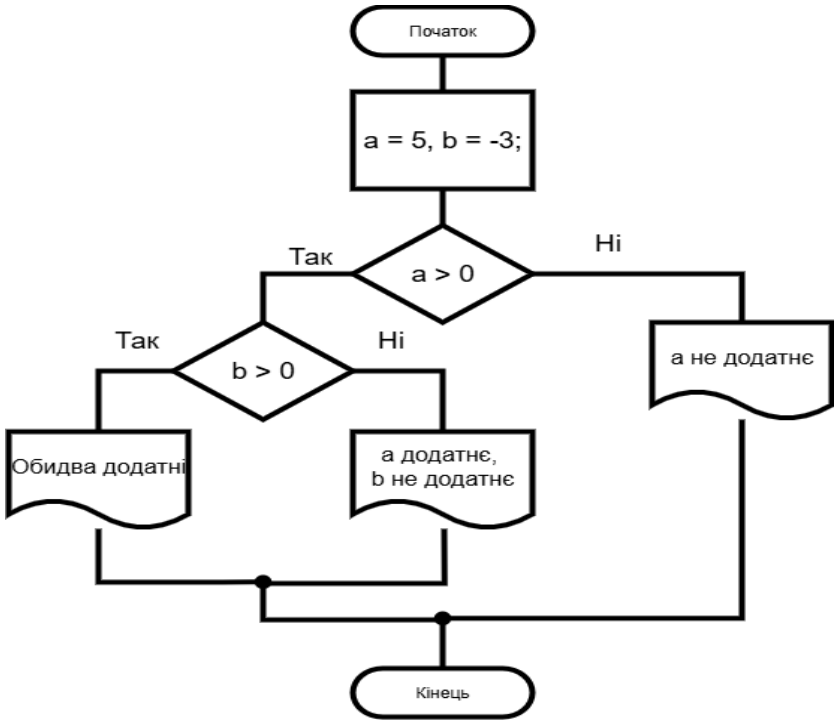


Рисунок 3.9 – Блок-схема алгоритму з вкладеним оператором `if`

Лістинг 3.1 – Приклад використання вкладених розгалужень

```
#include <stdio.h>

int main() {
    int a = 5, b = -3;

    if (a > 0) {
        if (b > 0) {
            printf("Обидва додатні\n");
        } else {
            printf("a додатне, b недодатне\n");
        }
    } else {
        printf("a недодатне\n");
    }
```

```
    return 0;
    }
```

Кінець лістингу 3.1

Тут спочатку перевіряється `a > 0.` Якщо істина – перевіряється друга умова `b > 0`. Таким чином маємо багаторівневий вибір.

Приклад вкладеного `switch` (рисунок 3.10, лістинг 3.2).

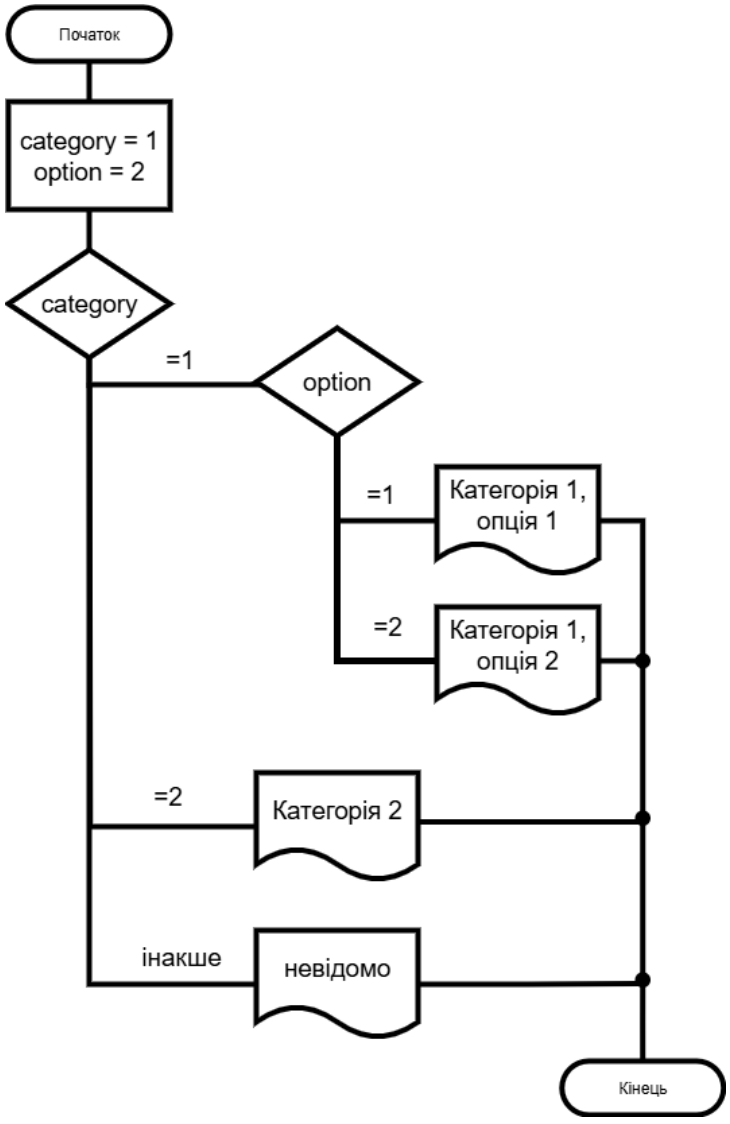


Рисунок 3.10 – Блок-схема алгоритму з вкладеним оператором `switch`

Лістинг 3.2 – Приклад використання оператора `switch`

```
#include <stdio.h>

int main() {
    int category = 1;
    int option = 2;
```

```
    switch (category) {
        case 1:
            switch (option) {
                case 1: printf("Категорія 1, опція 1\n");
break;
                case 2: printf("Категорія 1, опція 2\n");
break;
            }
            break;
        case 2:
            printf("Категорія 2\n");
            break;
        default:
            printf("Невідомо\n");
    }

    return 0;
    }
```

Кінець лістингу 3.2

Тут у випадку `category == 1` відбувається ще один вибір за допомогою вкладеного `switch`.

Отже, вкладені оператори розгалуження потрібні для *деталізованої логіки* з кількома рівнями умов. Але занадто багато вкладеностей роблять код важким для читання, краще іноді використовувати логічні вирази (`&&, ||`) або функції для спрощення.

3.6 Логічні вирази

Операндами логічних операцій є логічні дані, а саме:

– істина (`TRUE`) –довільне числове значення, яке не дорівнює 0;

– хибність (`FALSE`), що позначається 0.

Найчастіше операндами логічних операцій бувають вирази з операціями порівняння.

Операція логічного заперечення ! змінює логічне значення на протилежне. Вираз: `! (x > a && x < b)` буде істинним, коли `x ≠ (a, b)`. Цей вираз можна було б записати і так: `x<=a || x>=b.`

Операцію заперечення часто використовують у такому контексті: замість оператора порівняння `x == 0` записують `!x`.

Логічні вирази використовуються для перевірки умов і повертають істину (1) або хибність (0). Вони складаються з операндів і логічних операторів.

Основні логічні оператори наведені в таблиці 3.2.

Таблиця 3.2 – Логічні оператори

Оператор	**Назва**	**Приклад**	**Результат**
&&	логічне І (AND)	`(a > 0 && b > 0)`	істина, якщо обидві умови істинні
||	логічне АБО (OR)	`(a > 0 || b > 0)`	логічне АБО (OR)
!	логічне НЕ (NOT)	`!(a > 0)`	істина, якщо умова хибна

Приклад використання логічних операторів наведено в лістингу 3.3.

Лістинг 3.3 – Приклад використання логічних операторів

```
#include <stdio.h>

int main() {
    int a = 5, b = -3;
    if (a > 0 && b > 0) {
        printf("Обидва додатні\n");
    }

    if (a > 0 || b > 0) {
        printf("Хоча б одне додатне\n");
    }

    if (!(b > 0)) {
        printf("b не є додатним\n");
    }

    return 0;
    }
```

Кінець лістингу 3.3

Особливості:

– у виразі `a && b`, якщо `a = false` , то `b` навіть не обчислюється;

– у виразі `a || b`, якщо `a = true`, то `b` не обчислюється.

Логічні вирази часто включають оператори порівняння:

`==` (дорівнює);

`!=` (не дорівнює);

`> , <, >=, <=;`

Отже**,** логічні вирази – це поєднання умов і логічних операторів, результат яких використовується для прийняття рішень у програмах.

Контрольні питання

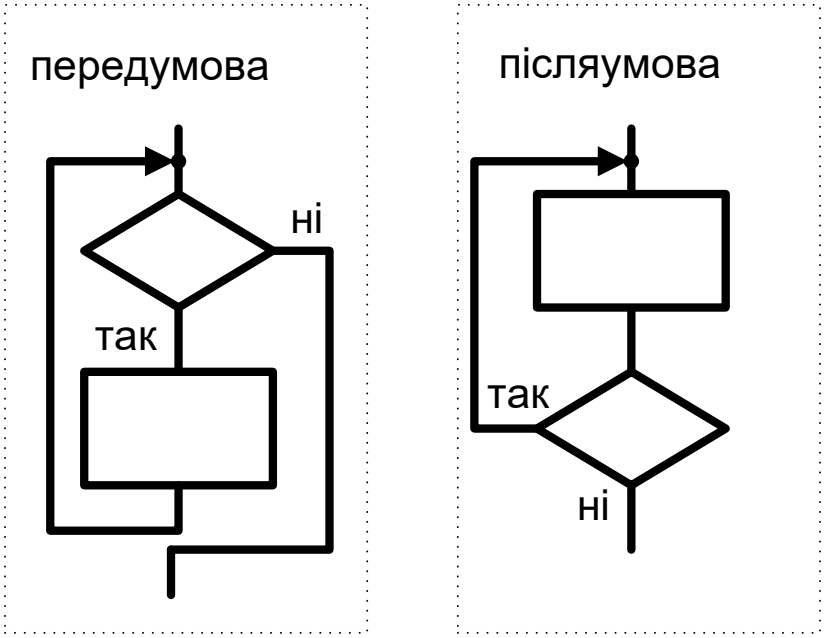
1. Що таке алгоритм і які існують основні способи його представлення?
2. У чому полягає відмінність між словесним, графічним та програмним способами подання алгоритму?
3. Що таке блок-схема алгоритму і які основні графічні елементи вона містить?
4. Яке призначення мають розгалуження в алгоритмах?
5. Який загальний синтаксис оператора розгалуження `if` у мові C/C++?
6. Як реалізується повна та неповна форма оператора `if`?
7. Як записується вкладений оператор `if` і в яких випадках доцільно його використовувати?
8. Що таке оператор вибору `switch` і коли його зручніше застосовувати замість `if`?
9. Які обмеження має оператор `switch` у мові C (щодо типів змінних і значень case)?
10. Що таке тернарний (умовний) оператор і як виглядає його синтаксис? Наведіть приклад.
11. У чому полягає різниця між оператором `if` і тернарним оператором?
12. Що таке логічний вираз і які логічні оператори використовуються в мові C?
13. Які значення мають істинність (`true`) і хибність (`false`) у мові C?
14. Як працюють логічні оператори `&&` (AND), `||` (OR) та `!` (NOT)?
15. Наведіть приклад програми, у якій використовується оператор розгалуження для знаходження більшого з двох чисел.

ТЕМА 4. Алгоритми, що містять повторення, оператори циклу

4.1 Циклічні алгоритми

Оператори циклу використовуються для організації обчислень, що *багато разів повторюються*.

Будь-який цикл складається з *тіла циклу*, тобто тих операторів, які виконуються кілька разів, *початкових установок*, *блоку модифікації параметра циклу* і перевірки *умови виходу з циклу* (рисунок 4.1), яка може розміщуватися або до тіла циклу (тоді говорять про циклю з передумовою), або після тіла циклу (цикл з післяумовою).



а) б)

Рисунок 4.1 – Види циклів

Один прохід циклу називається *ітерацією*. Змінні, умови виходу, що примусово змінюються в циклі і використовуються при перевірці, з нього, називаються *параметрами циклу*. Цілочисельні параметри циклу, що змінюються на ціле число на кожній ітерації, називаються *лічильниками циклу*.

Не можна передавати управління ззовні всередину циклу. Вихід з циклу можливий як при виконанні умови виходу, так і по операторах `break` або `continue.`

4.2 Оператор while

Існують задачі, коли заздалегідь невідомо скільки разів потрібно виконувати деякі оператори, але відома умова при який цикл виконується або умова при який цикл завершується. Коли спочатку потрібно перевірити умову, а потім виконувати тіло циклу використовується оператор циклу `while`.

Цикл з передумовою реалізує структурну схему, приведену на рисунку 4.1 (а), і має вигляд:

```
while ( вираз ) оператор;
```

Вираз визначає *умову повторення тіла циклу*, представленого простим або складеним оператором. Якщо вираз не рівний 0 (*істина*), виконується оператор циклу, після чого знову обчислюється вираз. Якщо при першій перевірці вираз рівний 0, цикл не виконається жодного разу. Тип виразу повинен бути арифметичним або таким, що приводиться до нього.

Приклад 4.1. Відобразити список цілих чисел від 1 до 5.

На перший погляд, цю задачу можна вирішити за допомогою єдиного виразу в C ++:

```
cout << "1\n2\n3\n4\n5\n";
```

Але що, якщо метою було б виведення списку з 50 чисел або 5000, тоді було б набагато ефективніше написати:

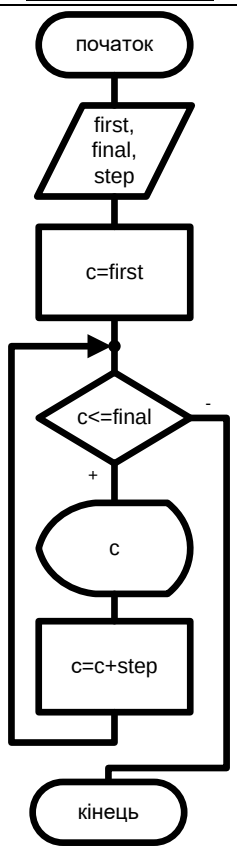
```
cout << N << endl;
```

де `N` – ціла змінна, яка використовується як лічильник. Лічильник може бути ініціалізований на 1 і регулярно збільшуватись на 1, доки не досягне найбільшого значення. Кожний прохід через операцію `cout` супроводжувався перевіркою умови, що лічильник був ще меншим, ніж максимальне (верхнє) значення (лістинг 4.1). Поки умова істинна (лічильник менший за праву межу),

тіло циклу повторюється. Але коли умова стане помилковою, здійснюється вихід з циклу і ми переходимо до наступного кроку програми (у цьому прикладі – в кінець).

Цикл з передумовою

Псевдокод	Блок-схема

BEGIN
get values of first & final & step
c ← first
WHILE c<=final
display c
c ← c+step
ENDWHILE
END
	

Лістинг 4.1 – Цикл з передумовою

```
#include <iostream> /*стандартна бібліотека вводу-виводу*/
using namespace std;

int c; /* Лічильник – число, що буде виводитися*/
int first; /* Початкове значення лічильника (ліва межа діапазону) */
int final; /* кінцеве значення лічильника (права межа діапазону) */
int step; /* крок зміни лічильника */

int main (void)
{
cout<<”first=”;
cin<<first;
cout<<”\nfinal=”;
cin<<final;
```

```
cout<<”\nstep=”;
cin<<step;
  c = first; //Ініціалізуємо лічильник початковим значенням
  while (c<=final)        /* Перевірка умови */
    {                     /* початок тіла циклу */
      cout << c << endl;  /* вивід числа */
      c = c + step;       /* зміна числа */
    }                     /* кінець тіла циклу */
  return 0;               /* повернути нульовий код помилки */
     }
```

Кінець лістингу 4.1

4.3 Оператор do while

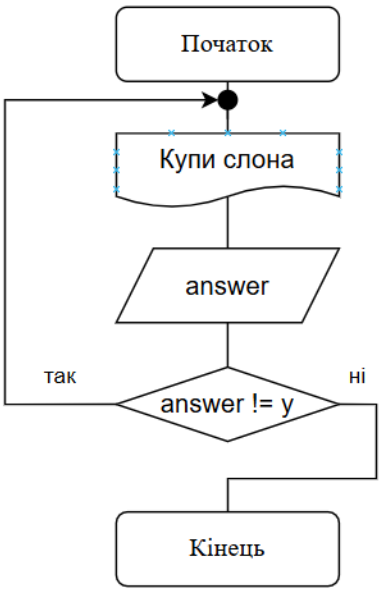
Цикл з післяумовою реалізує структурну схему, приведену на рисунку 4.1 (б), і має вигляд:

```
do оператор while вираз;
```

Спочатку виконується простій або складений оператор, що становить тіло циклу, а потім обчислюється вираз. Якщо умова істинна, тіло циклу виконується ще раз, і так далі, поки вираз не стане рівним нулю або в тілі циклу не буде виконаний який-небудь оператор передачі управління.

Приклад 4.2. Купи слона (лістинг 4.2).

Блок-схема



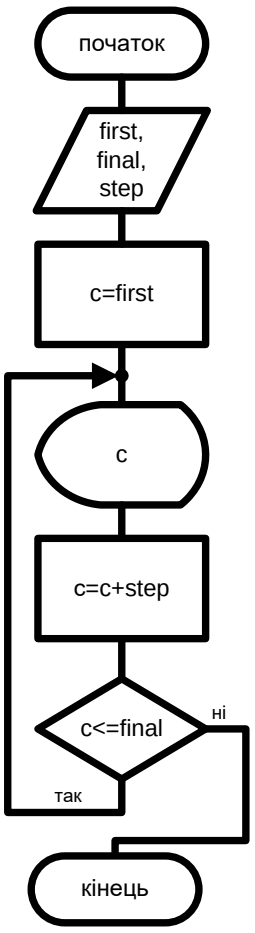
Лістинг 4.2 – Задача «Купи слона»

```
#include <iostream.h>
int main()
{
char answer;
do   {
     cout << "\nКупи слона! ";
cin >> answer;
}
while (answer != ‘y’);
     }
```

Кінець лістингу 4.2

Розв’яжемо задачу з прикладу 4.1 за допомогою циклу з післяумовою (лістинг 4.3).

Цикл з післяумовою

Псевдокод	**Блок-схема**
BEGIN get values of first & final & step c ← first **DO** display c c ← c+step **WHILE** (c<=final) **END**	

Лістинг 4.3 – Цикл з післяумовою

```
#include <iostream> // стандартна бібліотека вводу-виводу
using namespace std;
int c; /* Лічильник – число, що буде виводитися*/
int first; /* Початкове значення лічильника (ліва межа діапазону) */
int final; /* кінцеве значення лічильника (права межа діапазону) */
int step; /* крок зміни лічильника */

int main (void)
{
cout<<”first=”;
cin<<first;
cout<<”\nfinal=”;
cin<<final;
cout<<”\nstep=”;
cin<<step;
  c = first; /* Ініціалізуємо лічильник початковим значенням */
do
    {     /* початок тіла циклу */
      cout << c << endl;   /* вивід числа */
      c = c + step;        /* зміна числа */
    }                      /* кінець тіла циклу */
  while (c<=final);        /* перевірка умови */
  return 0;           /* повернення коду помилки 0*/
     }
```

Кінець лістингу 4.3

4.4 Цикл з параметром (for)

Цикл з параметром має наступний формат:

```
for ( ініціалізація; вираз; модифікації) оператор;
```

Ініціалізація використовується для оголошення і присвоєння початкових значень величинам, використовуваним в циклі. У цій частині можна записати декілька операторів, розділених комою.

Вираз визначає умову виконання циклу: якщо вона не рівне 0 (істинна), цикл виконується.

Модифікації виконуються після кожної ітерації циклу і служать зазвичай для зміни параметрів циклу. У частині модифікацій можна записати декілька операторів через кому.

Простий або складений оператор є *тілом циклу*. Будь-яка з частин оператора `for` може бути опущена (але крапки з комою треба залишити на своїх місцях!).

Цикл `for` часто використовується в програмі для тимчасових затримок, пов'язаних з необхідністю погодження швидкості реагування машини і можливістю сприйняття людиною:

```
for (n=1; n<=1000; n++);
```

Цей оператор примушує машину рахувати до `1000`, не виконуючи жодної дії, тому що в якості оператора тіла циклу стоїть порожній оператор (зверніть увагу на крапку з комою вкінці рядка).

В операторі `for` можуть бути реалізовані різні семантичні дії. Визначимо деякі з них:

1. Можна організувати ітерацію з від'ємним кроком:

```
for (n=10; n>0; n--)
        printf ("%d секунд! \n",n);
printf ("%d запуск! \n");
```

2. Можна реалізувати різний крок циклу:

```
for (n=5; n<60; n+=10)
    printf ("%d \n",n);
```

В цьому операторі значення n буде збільшуватись кожний раз на 10 і в результаті будуть надруковані числа 5, 15, 25, 35, 45, 55.

3. Можна вести підрахунок за допомогою символів, а не тільки чисел:

```
for (ch='a'; ch=='z'; ch++)
    printf ("Величина коду ASCII для %c дорівнює %d/
\n", ch, ch);
```

При виконанні цього оператору будуть друкуватись всі букви від `a` до `z` разом з їх кодами в `ASCII`. Цей оператор працює оскільки символи в

пам’яті машини розміщуються у вигляді чисел і тому в даному фрагменті, насправді, рахунок ведеться з використанням цілих чисел.

4. Можна перевірити виконання деякої специфічної умови, відмінної від умови, яка визначає число ітерацій (*але в такому випадку потрібно бути дуже уважним, щоб не вийшов нескінченний цикл!*):

```
for ( num=1; num*num*num<=216; num++)
```

Такий тип оператора `for` використовується у випадку, коли нас цікавить виконання якоїсь умови, а не кількість ітерацій (аналогічно оператору `while`).

5. В якості операції корекції параметру циклу може виступати не тільки операція додавання. Третім виразом в операторі `for` може використовуватись довільний правильний вираз, і його значення буде змінюватись на кожному кроку ітерації:

```
for (x=1; y<=75; y=5*x++ +10);
    printf (“%10d %10d \n”, x, y);
```

Зверніть увагу, що в специфікації перевіряється значення `y`, а не `x`. В кожному із трьох виразів оператору `for`, можуть використовуватись довільні змінні. Але *такий стиль програмування буде поганим*. Програма виглядає значно зрозумілішою. коли ми не змішуємо процес зміни параметру циклу з алгебраїчними обчисленнями.

6. Наявність кожного з трьох виразів не обов’язкова. Повинні бути присутніми символи “`;`” і оператори, які зрештою приведуть до закінчення роботи циклу:

```
ans=2;
for (n=3; ans<=25;)
ans*=n;
```

Тіло циклу

```
for (; ;)
    printf (“Я хочу щось зробити \n”);
```

буде виконуватись нескінченно, тому, що порожня умова завжди вважається істинною.

7. Перший вираз не повинен обов’язково ініціювати змінну. Але необхідно пам’ятати. що перший вираз виконується тільки один раз на початку роботи циклу:

```
for (printf (“Запам’ятовуйте введені числа! \n”); num==6;)
scanf (“%d”,& num);
```

В цьому фрагменті друкується повідомлення, а потім вводяться числа. Умовою закінчення введення є введення числа 6.

8. Параметри які вводять в три вирази специфікації циклу, можна змінювати при виконанні операції в тілі циклу.

Великий вибір вигляду виразів, які керують роботою циклу `for`, дозволяють за допомогою цієї конструкції робити значно більше, ніж просто організовувати ітераційний процес.

4.5 Використання операторів break і continue в операторах циклу

Оператор `break` може використовуватися в тілі циклу операторів `for, while, do while` для негайного виходу з оператора циклу. При цьому керування передається наступному за оператором циклу оператору, якщо цикл не є вкладеним в інший цикл. Якщо `break` використовується у складних циклічних процесах, то його дія поширюється лише на саму внутрішню структуру, в якій він безпосередньо перебуває.

Приклад використання оператора `break` наведено в лістингу 4.4.

Лістинг 4.4 – Приклад використання оператора `break`

```
#include <iostream>

void main ( )
{
for (int i = 1; i < 10; i++)
{
if (i = = 7) || db[il p wbrke ghb i = 7
break;
}
```

```
cout << endl << "При i рівному " << i << "цикл перерваний";
      }
```

Кінець лістингу 4.4

Оператор `continue` використовується в операторах також як оператор `break` в операторів `for, while, do while`, але його дія полягає не у виході з циклу, а в пропусканні частин операції, що залишились, і переході до початку наступної. Керування передається перевірочному виразу циклів `while`, `do while` або виразу керування циклом в `for`.

Приклад використання оператора `continue` наведено в лістингу 4.5.

Лістинг 4.5 – Приклад використання оператора `continue`

```
#include <iostream>
#define min 10
#define max 15
void main ( )
{
int i=0;
while (i < 20)
{
i+ = 1;
if (i > min && i < max)
continue;
cout << " i=" << i;
}
      }
```

Кінець лістингу 4.5

У результаті виконання програми на екран будуть виведені такі значення i: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 16, 17, 18, 19, 20.

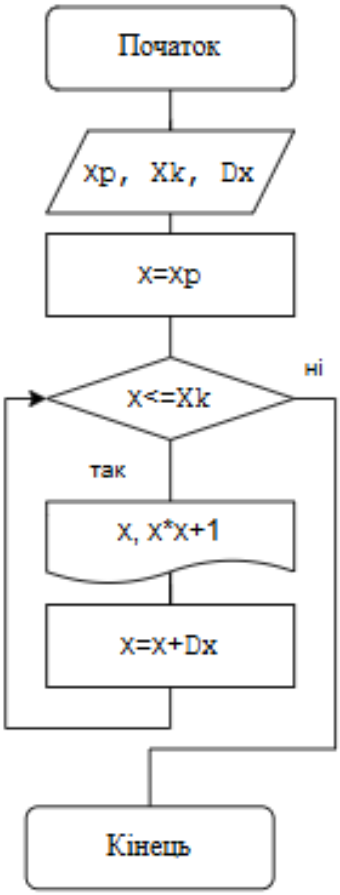
4.6 Розв'язування задач з використанням циклічних алгоритмів.

4.6.1 Табулювання функції

Приклад 4.3. Протабулювати функцію $y=x^2+1$ (надрукувати таблицю значень функції $y=x^2+1$ у введеному діапазоні).

Розв’яжемо цю задачу використавши цикл з передумовою (лістинг 4.6).

Блок-схема



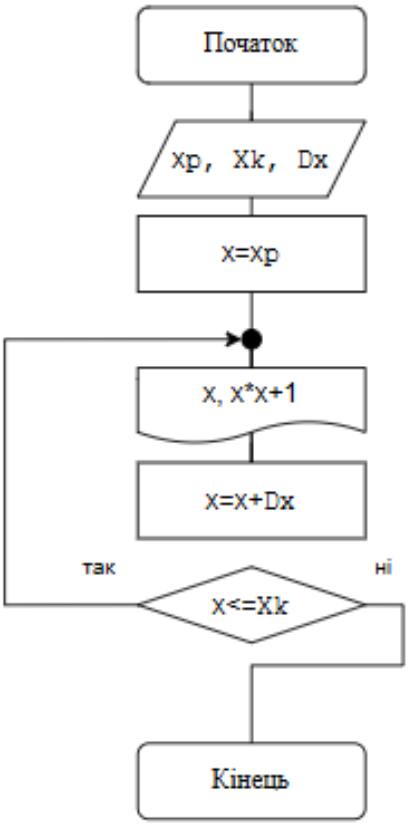
Лістинг 4.6 – Табулювання функції (цикл з передумовою)

```
#include <stdio.h>
int main()
{
     float Xp, Xk, Dx;
     printf("Введіть діапазон і крок зміни аргументу: ");
     scanf("%f%f%f" &Xp, &Xk, &Dx);
     printf("|   X   |   Y   |\n");//шапка таблиці
     float X = Xp;//початкові установки циклу
     while (X<=Xk){//перевірка умови виходу
     printf("| %5.2f | %5.2f |\n", X, X*X + 1);//тіло
     X += Dx;//модифікація
     }
      }
```

Кінець лістингу 4.6

Цю ж задачу можна розв’язати за допомогою циклу з післяумовою (лістинг 4.7).

Блок-схема



Лістинг 4.7 – Табулювання функції (цикл з післяумовою)

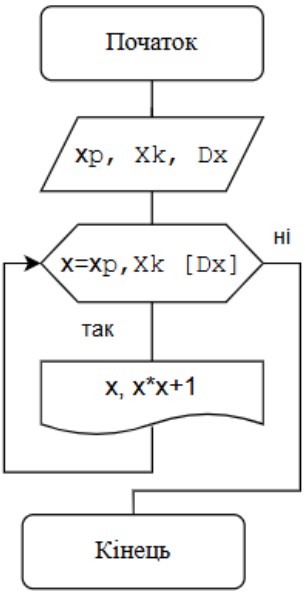
```
#include <stdio.h>
int main()
{
     float Xp, Xk, Dx;
     printf("Введіть діапазон і крок зміни аргументу: ");
     scanf("%f%f%f" &Xp, &Xk, &Dx);
     printf("|   X   |   Y   |\n");//шапка таблиці
     float X = Xp;//початкові установки циклу
     do
{
     printf("| %5.2f | %5.2f |\n", X, X*X + 1);//тіло
     X += Dx;//модифікація

     } while (X<=Xk){//перевірка умови виходу
    }
```

Кінець лістингу 4.7

Цю ж задачу можна розв’язати за допомогою циклу з параметром (лістинг 4.8).

Блок-схема



Лістинг 4.8 – Табулювання функції (цикл з параметром)

```
#include <stdio.h>
int main()
{
     float Xn, Xk, Dx;
     printf("Введіть діапазон і крок зміни аргументу: ");
     scanf("%f%f%f" &Xn, &Xk, &Dx);
     printf("|   X   |   Y   |\n");
     for (float X = Xn; X<=Xk; X += Dx)
               printf("| %5.2f | %5.2f |\n", X, X*X + 1);
    }
```

Кінець лістингу 4.8

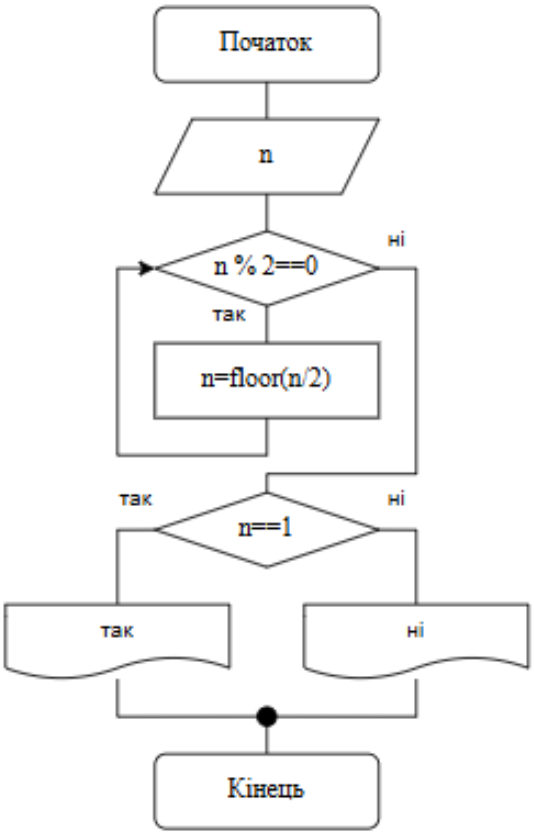
4.6.2 Вибір оптимального виду циклу для розв’язку задачі

Приклад 4.4. Дано натуральне число n. Чи є воно степенем числа 2?

Щоб з’ясувати, чи є число степенем числа 2, потрібно ділити це число на 2 (тобто виконувати оператор `n=floor(n/2)` поки це можна виконувати без остачі (тобто, якщо вірна умова `n % 2==0`). Тобто, спочатку ми перевіряємо умову – чи ділиться число `n` на `2` і, якщо ділиться, виконуємо ділення. Тому у цій задачі потрібно використовувати оператор циклу `while`.

В результаті одного чи декількох ділень числа на 2 ми отримаємо n=1, якщо число є степенем числа 2 (бо ділення на 2 завжди можливе), або n<>1, якщо число не є степенем числа 2 (лістинг 4.9).

Блок–схема



Лістинг 4.9 – Перевірка чи є число степенем числа 2

```
#include <iostream>
#include <math.h>
using namespace std;
int main()
{
    int n;
    cout << "введіть число n" << endl;
    cin >> n;
    while(n%2==0)
    {
        n=floor(n/2);
    }
    if (n==1)
    cout<<"Це число є степенем числа 2";
```

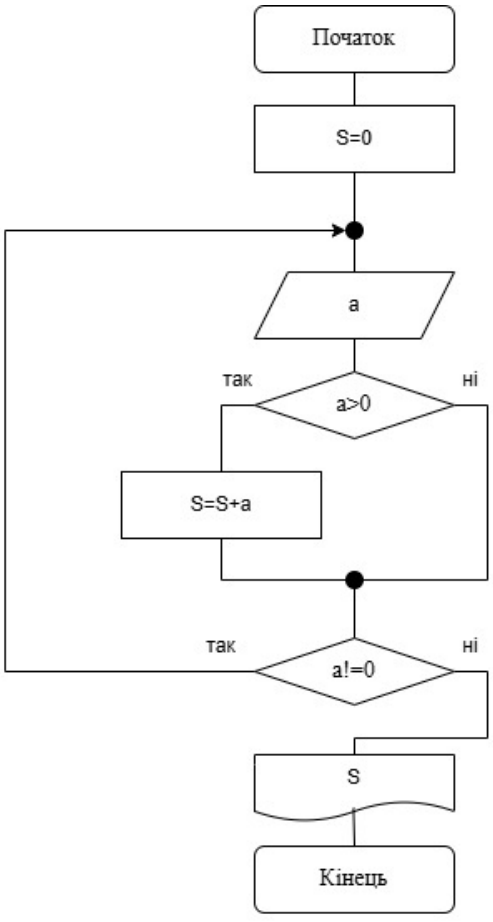
```
        else
        cout<<"Це число не є степенем числа 2";
    return 0;
    }
```

Кінець лістингу 4.9

Приклад 4.5. У циклі вводяться цілі числа. Умова закінчення вводу – ввід числа 0. Знайти суму додатних чисел.

Тобто, спочатку потрібно ввести число з клавіатури, зробити з ним потрібні дії, а потім перевірити і, якщо введене число рівне 0, то цикл потрібно завершити. Тому у цій задачі потрібно використовувати оператор циклу `do-while` (лістинг 4.10).

Блок-схема



Лістинг 4.10 – Перевірка чи є число степенем числа 2

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

```
int a, S=0;
do
{
cout << "введіть число " << endl;
cin >> a;
if(a>0) S+=a;
}
while(a!=0);
cout<<"S="<<S;
return 0;
}
```

Кінець лістингу 4.10

4.6.3 Пошук декількох чисел, що задовольняють деякій умові

Приклад 4.6. Знайдіть мінімальне число, яке більше за 200 та кратне 17.

Змінні:

Вхідних даних немає.

Вихідні: `n` –шукане число (цілого типу)

Алгоритм

1. Шукане число більше `200`. Тому пошук почнемо з числа `200`. Присвоїмо початкове значення змінній `n=200`. Оберемо цикл `do-while`, бо спочатку потрібно збільшити число `n` на `1`, а потім перевіряти умову завершення циклу `n % 17=0`.

2. У циклі будемо збільшувати число `n` на `1` (бо шукане число більше `200`) та перевіряти, чи воно кратне `17`.

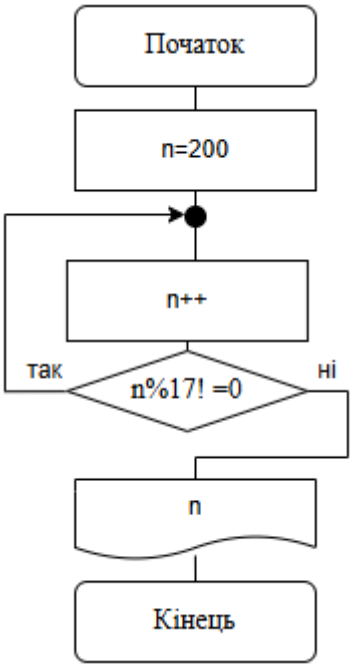
3. Якщо кратне, то кінець циклу.

4. Якщо не кратне перехід на новий виток циклу (на пункт 2).

5. Цикл завершується, коли число `n % 17=0`.

6. Після завершення циклу вивід знайденого числа на екран (лістинг 4.11).

Блок-схема



Лістинг 4.11 – Знаходження числа, кратного 17

```
#include <iostream>
using namespace std;
int main()
{
    int n=200;
    do
    {
    n++;
    }
    while(n%17!=0);
    cout<<"n="<<n;
    return 0;
    }
```

Кінець лістингу 4.11

Приклад 4.7. Знайдіть 5 перших чисел, що більші за 200 та кратні 17.

Змінні:

Вхідних даних немає.

Вихідні: n –шукані числа (цілого типу).

Проміжні: k – лічильник шуканих чисел (цілого типу).

Алгоритм

1. Шукані числа більші за `200`. Тому пошук почнемо з числа `200`. Присвоїмо початкове значення змінній `n=200`, а змінній `k=0`. Також оберемо цикл `do-while`.

2. У циклі будемо:

– збільшувати число n на 1 (бо шукані числа більші за `200`);

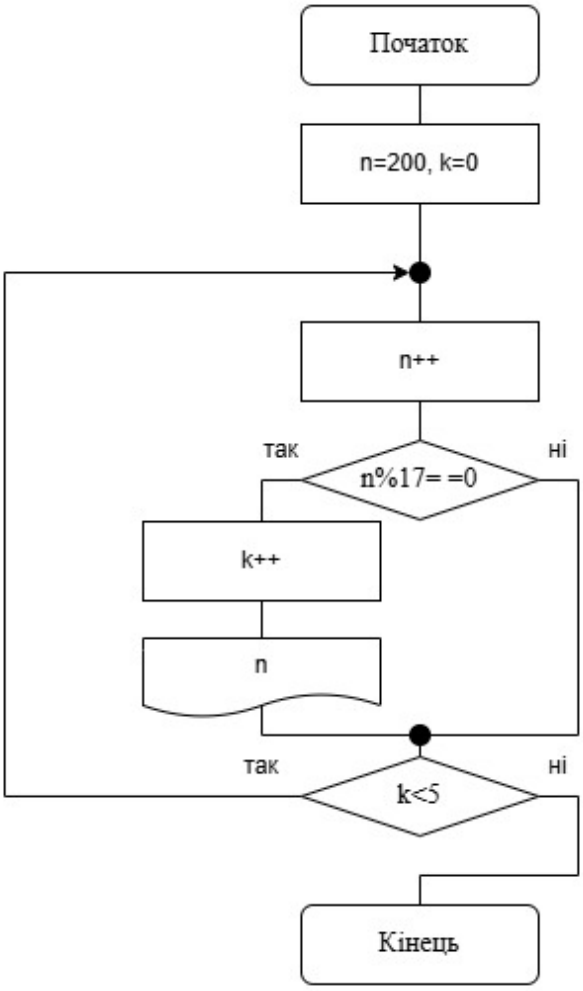
– перевіряти, чи воно кратне `17`;

– якщо кратне, то надрукуємо це число та збільшимо лічильник чисел k на `1`;

– якщо не кратне перехід на новий виток циклу (на пункт 2);

– цикл завершується, коли надруковано вже `5` чисел, тобто якщо `k=5` (лістинг 4.12).

Блок-схема



Лістинг 4.12 – Знаходження 5 чисел, кратних 17

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

int main()
{
    int n=200,k=0;
    cout<<"5 чисел, які більші за 200 та кратні 17:\n";

    do
    {
    n++;
    if(n%17==0)
        {
        k++;
        cout<<n<<"\t";
        }
    }
    while(k<5);

    return 0;
    }
```

Кінець лістингу 4.12

4.6.4 Задачі на перебір цифр натурального числа

Описаний нижче стандартний алгоритм дозволяє виконувати будь-які дії з цифрами натурального числа, в незалежності від значності числа.

Число, після використання цього алгоритму, стає рівним 0. Тому рекомендується зберігати вхідне значення числа у іншій змінній.

Приклад 4.8. Дано натуральне число n. Визначте у ньому кількість та суму цифр.

Змінні:

Вхідні: `n` – натуральне число (цілого типу `longint`).

Вихідні: `k` –кількість цифр числа (цілого типу `byte`); `s` – сума цифр числа (цілого типу `byte`).

Проміжні: `c` – остання цифра числа (цілого типу `byte`).

Алгоритм

Спочатку потрібно ввести число.

Нам потрібно знайти суму та кількість. Тому встановимо початкове значення змінним `k` та `s`, присвоїмо їм `0`.

Згадаємо формулу знаходження останньої цифри числа: `c= n%10`. Ця формула вірна для числа будь-якої значності.

Згадаємо, що після виконання оператору `n=floor(n/10)`, від числа відкидається остання цифра.

На цих двох формулах заснований такий алгоритм: обчислюємо останню цифру числа, робимо з нею потрібні дії (знаходимо кількість, суму, найбільшу цифру, і.т.д), потім цифру відкидаємо. Це робимо, поки у введеному числі не залишиться цифр, тобто число стане 0. Ясно, що для такого алгоритму потрібен цикл з післяумовою do-while. Докладніше, у цьому циклі будемо виконувати такі дії:

Оператор `c= n % 10` обчислює останню цифру числа.

Оператор `k=k+1` збільшує лічильник цифр числа.

Оператор `s=s+c` накопичує знайдену цифру у суму.

Оператор `n= floor(n/10)`, відкидає останню цифру числа. Цей оператор дуже важливий, бо він впливає на умову. Якщо його не написати, програма зациклиться.

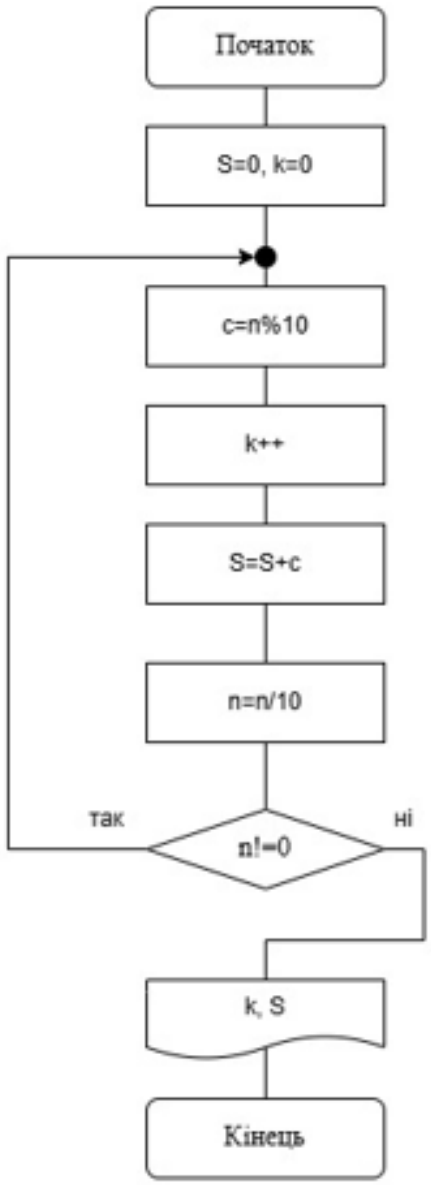
Перевіряється умова `n=0` – чи є ще цифри у числі?

Якщо умова не вірна, тобто цифри у числі є, то виконується перехід на початок циклу.

Якщо умова вірна, тобто цифр у числі немає, то цикл завершується і виконується перехід на оператор, що іде після циклу.

Коли цикл закінчиться, тобто будуть видалені всі цифри числа `n` виводимо на екран знайдені кількість та суму цифр `k, s` (лістинг 4.13).

Блок–схема



Лістинг 4.13 – Знаходження кількості та суми цифр в числі

```
#include <iostream>
using namespace std;
int main()
{
int n;
int S=0,k=0;
int c;
cout<<"Введіть число ";
cin>>n;
    do
    {
    c=n%10;
```

```
    k++;
    S+=c;
    n=n/10;
    }
    while(n>0);
cout<<"В цьому числі "<<k<<" цифр, їх сума становить "<<S;
    return 0;
    }
```

Кінець лістингу 4.13

4.6.5 Задачі на зміну цифр натурального числа

Згадаємо, що десяткова система числення є позиційною. В позиційній системі числення значення кожної цифри залежить від її позиції – місця у числі.

Число 4131 насправді має таки склад:

$$4131=4*1000+1*100+3*10+1=4*10^3+1*10^2+3*10^1+1*10^0$$

Число 10, степені якого використовуються у цій формулі називається основою системи числення, а степені десятки – це вага цифри.

Приклад 4.9. Дано натуральне число. Змініть у ньому всі цифри 1 на 2.

Змінні:

Вхідні: `n` – натуральне число (цілого типу `longint`).

Вихідні: `m` – нове натуральне число, в якому замість `1` стоять `2` (цілого типу `longint`).

Проміжні: `c` – остання цифра числа (цілого типу `byte`); `d` – вага цифри у числі – 1, 10, 100, 1000, ... (цілого типу `longint`).

Алгоритм

Скорочений алгоритм формування з введеного числа `n` нового числа `m` такий:

Знаходимо `c` – останню цифру числа `n`.

Якщо ця цифра `1`, змінюємо її на `2`, якщо ні, то не змінюємо;

Додаємо цю цифру `c` в початок нового числа `m` (враховуючи вагу `d`).

Відкидаємо цифру з числа `n`.

Все це повторюємо, поки не закінчаться цифри у числі n.

Щоб визначити формули, за якими будуть обчислюватись m та d складемо таблицю для нашого прикладу (таблиця 4.1).

Таблиця 4.1 – Визначення формул для алгоритму розв'язування задачі прикладу 4.9

Змінна	***c***	***m***	***d***	***n***
Дія	*Обчислення останньої цифри*	*Накопичуємо цифру у початок числа*	*Вага цифри, степені числа 10*	*Відкидаємо останню цифру*
Початкове значення		0	1	4131
1 виток циклу	1 змінюємо на 2	2	10	413
2 виток циклу	3	32=3*10+2	100	41
3 виток циклу	1 змінюємо на 2	232=2*100+32	1000	4
Останній виток циклу	4	4232=4*1000+232	10000	0
Формули	c=n % 10	m = c*d + m	d=d*10	n= floor(n/10)

Аналізуючи третій стовпчик таблиці можна заповнити комірку, де буде формула для обчислення нового числа m. Ясно, що для додавання цифри c у початок нового числа m потрібно помножати її на вагу d і цей добуток накопичувати у нове число m.

Формула для обчислення степенів числа 10 нам відома. Можна записати її у четвертому стовпчику.

Розглянемо алгоритм докладніше.

Спочатку потрібно ввести число.

Нам потрібно сформувати нове число m. Ми будемо його накопичувати, як суму, тому потрібно встановити початкове значення m=0.

Ми будемо використовувати d вагу цифри у числі. Ми будемо її накопичувати, як добуток, тому потрібно встановити початкове значення d=1.

У вже відомий нам алгоритм перебору цифр числа, додамо два оператори (обчислення m та d), які умовно можна назвати «формування нового числа». Докладніше, у циклі будемо виконувати такі дії:

Оператор c= n % 10 обчислює останню цифру числа.

Оператор if (c= =1) c=2 змінює цифру 1 на 2.

Оператор `m=m+c*d` додає знайдену цифру у початок нового числа, враховуючі її вагу.

Оператор `d=d*10` обчислює вагу наступної цифри.

Оператор `n= floor (n/10)` відкидає останню цифру числа.

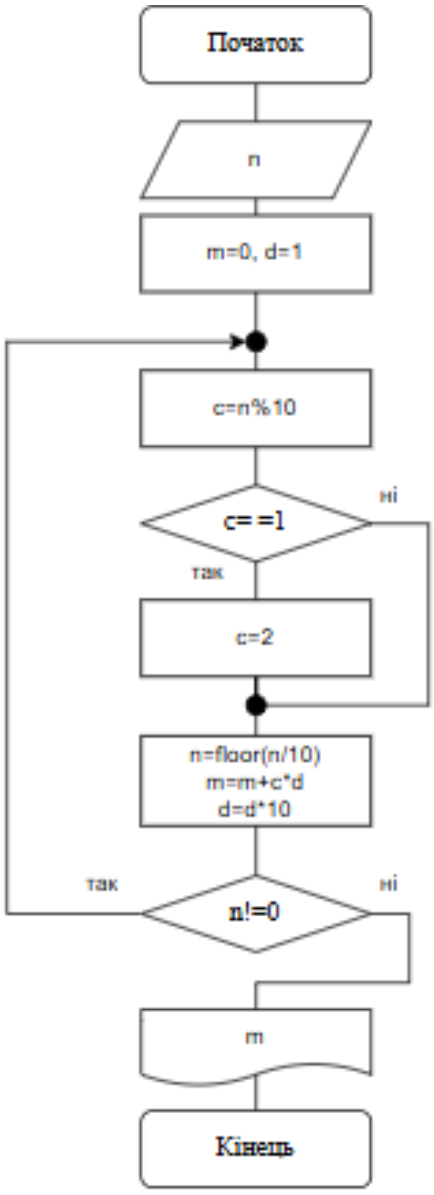
Перевіряється умова `n!=0` – чи є ще цифри у числі.

Якщо умова вірна, тобто цифри у числі є, то виконується перехід на початок циклу.

Якщо умова не вірна, тобто цифр у числі немає, то цикл завершується і виконується перехід на оператор, що іде після циклу.

Коли цикл закінчиться, тобто будуть видалені всі цифри числа n виводимо на екран знайдене нове число `m` (лістинг 4.14).

Блок–схема



Лістинг 4.14 – Заміна цифр в числі

```
#include <iostream>
using namespace std;
int main()
{
int n;
int m=0,d=1;
int c;
cout<<"Введіть число ";
cin>>n;
    do
    {
    c=n%10;
    if (c==1) c=2;
    n=n/10;
    m=m+c*d;
    d=d*10;
    }
    while(n!=0);
cout<<"m="<<m;
    return 0;
    }
```

Кінець лістингу 4.14

Приклад 4.10. Дано натуральне число n. Вилучить у ньому всі 1.

Змінні:

Вхідні: `n` – натуральне число (цілого типу `longint`).

Вихідні: `m` – нове натуральне число, в якому немає `1` (цілого типу `longint`).

Проміжні: `c` – остання цифра числа (цілого типу `byte`); `d` – вага цифри у числі – 1, 10, 100, 1000, ... (цілого типу `longint`).

Алгоритм

Алгоритм формування з введеного числа `n` нового числа `m` без `1` відрізняється від алгоритму попередньої задачі тим, що оператори, які ми назвали «формування нового числа» виконуються тільки, коли цифра, яка переноситься у нове число не `1`. Якщо цифра дорівнює `1`, то ці оператори не виконуються і цифра в нове число не переноситься.

Розглянемо алгоритм докладніше.

Спочатку потрібно ввести число.

Встановимо початкове значення нового числа `m=0`.

Встановимо початкове значення ваги `d=1`.

У циклі будемо виконувати такі дії:

Оператор `c= n % 10` обчислює останню цифру числа.

Якщо ця цифра не `1` (оператор `if c<>1`),то переносимо її у нове число.

Оператор `m=m+c*d` додає знайдену цифру у початок нового числа, враховуючі її вагу.

Оператор `d=d*10` обчислює вагу наступної цифри.

Оператор `n= floor (n/10)` відкидає останню цифру числа `n`.

Перевіряється умова `n!=0` – чиє ще цифри у числі.

Якщо умова не вірна, тобто цифри у числі є, то виконується перехід на початок циклу.

Якщо умова вірна, тобто цифру числі немає, то цикл завершується і виконується перехід на оператор, що іде після циклу.

Коли цикл закінчиться, тобто будуть видалені всі цифри числа n виводимо на екран знайдене нове число.

Приклад 4.11. Дано натуральне число n. Переверніть число. Наприклад, з числа 3456 одержати 6543.

Змінні:

Вхідні: `n` – натуральне число (цілого типу `longint`).

Вихідні: `m` – нове перевернуте натуральне число (цілого типу `longint`).

Проміжні: `c` – остання цифра числа (цілого типу `byte`).

Алгоритм

Скорочений алгоритм формування з введеного числа `n` нового числа `m` такий:

Знаходимо `c` – останню цифру числа `n`.

Додаємо цю цифру `c` у кінець нового числа `m` (для додавання цифри у кінець числа, потрібно число помножити на `10` та додати цифру).

Відкидаємо цифру з числа `n`.

Все це повторюємо, поки не закінчаться цифри у числі n.

Щоб визначити формулу, за якою буде обчислюватись m складемо таблицю для нашого прикладу (таблиця 4.2).

Таблиця 4.2 – Таблиця 4.1 – Визначення формул для алгоритму розв'язування задачі прикладу 4.11

Змінна	***c***	***m***	***n***
Дія	*Обчислення останньої цифри*	*Накопичуємо цифру у початок числа*	*Відкидаємо останню цифру*
Початкове значення		0	3456
1 виток циклу	6	6	345
2 виток циклу	5	65=6*10+5	34
3 виток циклу	4	654=65*10+4	3
Останній виток циклу	3	6543=654*10+3	0
Формули	c=n % 10	m= m*10+c	n= floor (n/10)

Аналізуючи третій стовпчик таблиці можна заповнити комірку, де буде формула для обчислення нового числа m. Ясно що для додавання цифри c у кінець нового числа m потрібно число m помножати на 10 та додавати да нього знайдену цифру c.

Розглянемо алгоритм докладніше.

Спочатку потрібно ввести число

Нам потрібно сформувати нове число m. Ми будемо його накопичувати, як суму, тому потрібно встановити початкове значення m=0.

У цьому прикладі у відомий нам алгоритм перебору цифр числа, додамо один оператор (обчислення m), який умовно можна назвати «формування перевернутого числа». Докладніше, у циклі будемо виконувати такі дії:

Оператор c= n % 10 обчислює останню цифру числа.

Оператор m=m*10+c додає знайдену цифру у кінець нового числа.

Оператор n= floor (n/10) відкидає останню цифру числа.

Перевіряється умова n!=0 – чи є ще цифри у числі.

Якщо умова не вірна, тобто цифри у числі є, то виконується перехід на початок циклу.

Якщо умова вірна, тобто цифр у числі немає, то цикл завершується і виконується перехід на оператор, що іде після циклу.

Коли цикл закінчиться, тобто будуть видалені всі цифри числа `n` виводимо на екран знайдене нове число.

Контрольні питання

1. Що таке циклічний алгоритм і в яких випадках його доцільно використовувати?
2. Які види циклів підтримуються мовою програмування С?
3. У чому полягає відмінність між циклами з передумовою і з післяумовою?
4. Який загальний синтаксис оператора циклу `while` у мові С?
5. Як працює цикл `while` і коли він завершує виконання?
6. У чому полягає особливість виконання циклу `do…while`?
7. У яких випадках доцільно використовувати цикл `do…while` замість `while`?
8. Який загальний синтаксис циклу з параметром `for` у мові С?
9. Які три частини містить оператор `for` і яку роль виконує кожна з них?
10. Наведіть приклад використання циклу `for` для обчислення суми чисел від 1 до 100.
11. Як можна організувати вкладені цикли та які особливості їх виконання?

ТЕМА 5. Масиви

5.1 Поняття масиву даних

Проста статична структура даних – масив, де звернення до елементу відбувається через його номер.

Слово «масив» вживається в різних контекстах:

– як множина з операціями: одержати елемент з номером N, записати елемент з номером N;

– як фізична структура, реалізована у вигляді неперервної області пам’яті.

У разі реалізації масиву через таку структуру(мови Паскаль, C/C++) номер відповідає зсуву від початку області. У деяких мовах, наприклад, PHP, масив реалізований по-іншому.

Масив – це структура даних, що являє собою однорідну (за типом), фіксовану (за розміром і конфігурацією) сукупність елементів, упорядкованих за номерами. Масив визначається ім’ям (ідентифікатором) і кількістю індексів (номерів), що потрібні для визначення місцезнаходження необхідного елементу масиву. Ім’я масиву є єдиним для всіх його елементів.

В програмуванні кількість індексів масиву називають його *розмірністю*, кількість дозволених значень кожного індексу – його *діапазоном*, а сукупність розмірності та діапазону – *формою масиву*.

Масиви використовуються для зберігання кількох значень в одній змінній замість оголошення окремих змінних для кожного значення.

Порядок роботи з масивом:

1) оголосити про масив у розділі описів, вказавши його розмір і тип елементів, що в нього входять (тобто приготувати місце в пам’яті, де будуть зберігатися значення елементів);

2) заповнити необхідними значеннями масив для розв’язування задачі;

3) якщо треба, вивести масив на екран для перевірки коректності роботи;

4) робота з даним масивом;

5) виведення отриманих результатів.

Під час розв’язування задач, як правило, використовуються одновимірні та двовимірні масиви. Масиви більшої розмірності на практиці майже не зустрічаються.

5.2 Одновимірний масив

Одновимірний масив інакше ще називають лінійним масивом або *вектором*. Кожному його елементу ставиться у відповідність один індекс (рисунок 5.1). Для роботи з масивом необхідно використовувати будь-який оператор повторення (циклу), тому що кожна дія з його елементами виконується однаково.

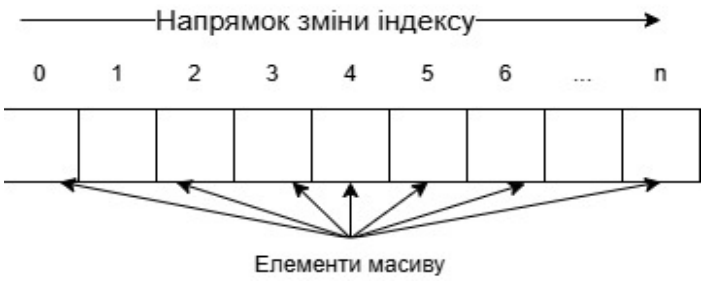


Рисунок 5.1 – Індекси елементів масиву

5.2.1 Оголошення масиву та доступ до його елементів

Щоб оголосити масив, визначте тип змінної, вкажіть назву масиву, а потім у *квадратних дужках* вкажіть кількість елементів, які він має зберігати:

```
string cars[4];
```

Ми оголосили змінну, яка містить масив із чотирьох рядків. Щоб заповнити його значеннями, ми можемо використати літерал масиву – розмістити значення у списку, розділеному комами, у *фігурних дужках*:

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
```

Щоб створити масив із трьох цілих чисел, можна написати:

```
int myNum[3] = {10, 20, 30};
```

У C++ можемо не вказувати розмір масиву. Компілятор достатньо розумний, щоб визначити розмір масиву на основі кількості вставлених значень:

```
string cars[] = {"Volvo", "BMW", "Ford"};
// Три елементи масиву
```

Аналогічно можемо писати:

```
string cars[3] = {"Volvo", "BMW", "Ford"};
// Також три елементи масиву
```

Однак останній підхід вважається «хорошою практикою», оскільки він зменшить ймовірність помилок у нашій програмі.

Також можна оголосити масив, не вказуючи елементи в оголошенні, і додати їх пізніше:

```
string cars[5];
cars[0] = "Volvo";
cars[1] = "BMW";
cars[2] = "Ford";
cars[3] = "Mazda";
cars[4] = "Tesla";
```

Проте, наведений вище приклад працює лише тоді, коли ми вказали розмір масиву.

Щоб отримати розмір масиву, можна використати оператор `sizeof()` (рисунок 5.2).

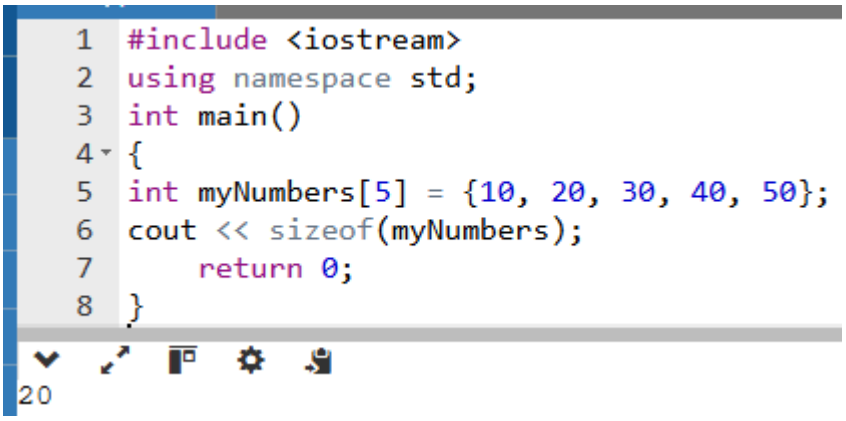


Рисунок 5.2 – Знаходження розміру масиву

Чому в результаті показано `20` замість `5`, коли масив містить `5` елементів?

Це тому, що оператор `sizeof()` повертає розмір типу в байтах.

Тип `int` зазвичай має 4 байти, тому з прикладу вище, 4 x 5 (4 байти x 5 елементів) = 20 байтів.

Щоб дізнатися, скільки елементів містить масив, потрібно розділити розмір масиву на розмір типу даних, який він містить:

```
int getArrayLength = sizeof(myNumbers) / sizeof(int);
```

Результат використання такого синтаксису можна подивитися на рисунку 5.3.

main.cpp

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int myNumbers[5] = {10, 20, 30, 40, 50};
6  int getArrayLength = sizeof(myNumbers) / sizeof(int);
7  cout << getArrayLength;
8  return 0;
9  }
```

```
5
```

Рисунок 5.3 – Знаходження розміру масиву

5.2.2 Доступ до елементів одновимірного масиву

Масив розміщується в оперативній пам’яті, починаючи з деякої адреси, що іменується базою, а решта елементів знаходиться на певній відстані від бази – зсуві. Для всіх елементів виділяється однакове місце, що визначається їх типом.

Щоб отримати доступ до елемента масиву, потрібно вказати номер його індексу в квадратних дужках `[]`.

Приклад. Дано масив з іменем A з 10 цілих чисел. Записати число 5 в перший та останній елементи масиву.

```
int A[10]; // опис масиву A
A[0] = 5; // перший елемент масиву
```

```
A[9] = 5; // останній елемент масиву
```

Ілюстрація коду, наведеного вище, на рисунку 5.4.

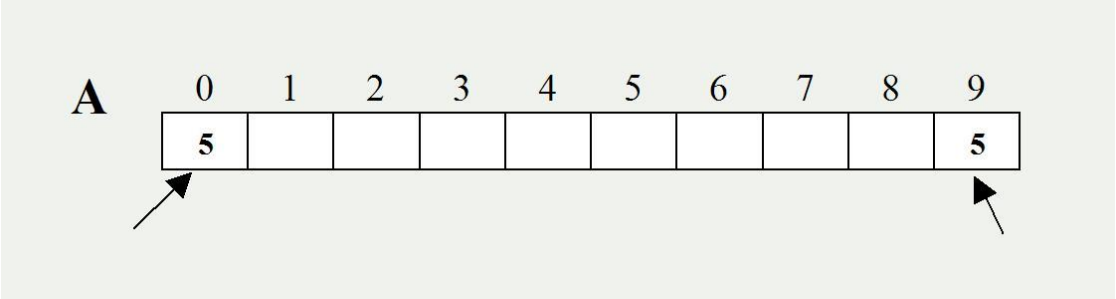


Рисунок 5.4 – Доступ до елементів масиву

Примітка. Індекси масиву починаються з 0: `[0]` є першим елементом, `[1]` – другий елемент і т.д.

Ця інструкція отримує доступ до значення першого елемента в `cars`:

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cout << cars[0];// Виведе Volvo
```

Щоб змінити значення певного елемента, зверніться до номера індексу:

```
cars[0] = "Opel";
```

Приклад

```
string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};
cars[0] = "Opel";
cout << cars[0];// Тепер виведе Opel замість Volvo
```

5.2.3 Методи заповнення одновимірного масиву

1) за формулою:

```
for (i=1; i<10;i++)
M[i]=i*i-10 {або будь-яка формула};
```

2) з клавіатури:

```
for (i=1; i<10;i++)
{
cout<< ‘Введіть M[‘<<i<<‘]: ‘);
cin>>M[i];
}
```

3) випадковим чином (генератором випадкових чисел) із проміжку [A, B]:

```
for (i=1; i<10;i++)
M[i]=random(B-A)+A;
```

5.2.4 Методи виведення елементів одновимірного масиву на екран

1) виведення у стовпчик наведено на рисунку 5.5.

main.cpp

```
#include <iostream>
using namespace std;
//оголошення масиву
float M[10];
int i;
int main()
{
//заповнення масиву
for (i=0; i<10;i++)
M[i]=i*i-10;
//виведення масиву у стовпчик
for (i=0; i<10;i++)
cout<<M[i]<<endl;
    return 0;
}
```

```
-10
-9
-6
-1
6
15
26
39
54
71
```

Рисунок 5.5 – Виведення одновимірного масиву в стовпчик

2) виведення у рядок наведено на рисунку 5.6.

main.cpp

```
#include <iostream>
using namespace std;
//оголошення масиву
float M[10];
int i;
int main()
{
//заповнення масиву
for (i=0; i<10;i++)
M[i]=i*i-10;
//виведення масиву у рядок
for (i=0; i<10;i++)
cout<<M[i]<<" ";
    return 0;
}
```

```
-10 -9 -6 -1 6 15 26 39 54 71
```

Рисунок 5.6 – Виведення одновимірного масиву в рядок

Дозволяється об’єднувати в одному циклі кілька етапів розв’язування задачі. Наприклад, очищення, заповнення масиву та виведення елементів

масиву для контролю на екран.

Подивіться на два приклади коду нижче і подумайте в чому різниця та яким буде результат:

Приклад 1

```
string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
for (int i = 0; i < 5; i++) {
 cout << cars[i] << "\n";
}
```

Приклад 2

```
string cars[5] = {"Volvo", "BMW", "Ford", "Mazda", "Tesla"};
for (int i = 0; i < 5; i++) {
 cout << i << " = " << cars[i] << "\n";
}
```

Використовуючи підхід `sizeof()`, можемо створювати цикли, які працюють для масивів будь-якого розміру, що є більш стійким.

Замість того, щоб писати:

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i++) {
  cout << myNumbers[i] << "\n";
}
```

Краще написати:

```
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < sizeof(myNumbers) / sizeof(int); i++) {
  cout << myNumbers[i] << "\n";
}
```

5.3 Багатовимірні масиви

5.3.1 Поняття багатовимірного масиву

Багатовимірний масив – це масив масивів.

Щоб оголосити багатовимірний масив, визначте тип змінної, вкажіть ім’я масиву, а потім квадратні дужки, які вказують, скільки елементів містить основний масив, а потім інші квадратні дужки, які вказують, скільки елементів мають підмасиви:

```
string letters[2][4];
```

Як і у звичайних масивах, ви можете вставляти значення за допомогою літералу масиву – списку, розділеного комами, у фігурних дужках. У багатовимірному масиві кожен елемент літералу масиву є іншим літералом масиву.

```
string letters[2][4] = {
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};
```

Кожен набір квадратних дужок в декларації масиву додає ще один *вимір* до масиву. Кажуть, що такий масив, як наведений вище, має два виміри.

Масиви можуть мати будь-яку кількість розмірів. Чим більше розмірів має масив, тим складнішим стає код. Наступний масив має три виміри:

```
string letters[2][2][2] = {
  {
    { "A", "B" },
    { "C", "D" }
  },
  {
    { "E", "F" },
    { "G", "H" }
  }
};
```

5.3.2 Доступ до елементів двовимірного масиву

Приклад двовимірного масиву та його індексації наведено на рисунку 5.7.

Рисунок 5.7 – Приклад двовимірного масиву

Адресація елементів масиву Mas2 «дорожча» – тепер вона вимагає вказівки двох індексів, а саме, зовнішнього (рядок) і внутрішнього – природно назвати його стовпцем.

Звернення до елементів двовимірного масиву відбувається за допомогою двох індексів, проте в пам’яті такий масив зберігається послідовно (рисунок 5.8).

Рисунок 5.8 – Індексація двовимірного масиву

Щоб отримати доступ до елемента багатовимірного масиву, вкажіть номер індексу в кожному вимірі масиву.

Ця інструкція отримує доступ до значення елемента в першому рядку (0) і третьому стовпці (2) масиву `letters` (літери):

```
string letters[2][4] = {
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};
cout << letters[0][2];  // Виведе "C"
```

5.3.3 Зміна елементів у багатовимірному масиві

Щоб змінити значення елемента, зверніться до номера індексу елемента в кожному з вимірів:

```
string letters[2][4] = {
```

```
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};
letters[0][0] = "Z";
cout << letters[0][0];  // Тепер виведе "Z" замість "A"
```

Щоб виконати цикл багатовимірного масиву, вам потрібен один цикл для кожного виміру масиву.

5.3.4 Вивід елементів багатовимірного масиву

У наступному прикладі виводяться всі елементи в масиві `letters`:

```
string letters[2][4] = {
  { "A", "B", "C", "D" },
  { "E", "F", "G", "H" }
};

for (int i = 0; i < 2; i++) {
  for (int j = 0; j < 4; j++) {
    cout << letters[i][j] << "\n";
  }
}
```

Подібним чином можна опрацьовувати і тривимірний масив:

```
string letters[2][2][2] = {
  {
    { "A", "B" },
    { "C", "D" }
  },
  {
    { "E", "F" },
    { "G", "H" }
  }
};

for (int i = 0; i < 2; i++) {
  for (int j = 0; j < 2; j++) {
    for (int k = 0; k < 2; k++) {
      cout << letters[i][j][k] << "\n";
    }
  }
}
```

Багатовимірні масиви чудово відображають сітки. Цей приклад показує їх практичне використання. У наступному прикладі використано багатовимірний масив для представлення невеличкої гри Battleship:

```
// Використовуємо 1 щоб вказати, що є корабель
bool ships[4][4] = {
  { 0, 1, 1, 0 },
  { 0, 0, 0, 0 },
  { 0, 0, 1, 0 },
  { 0, 0, 1, 0 }
};

// Слідкуємо за тим, скільки ударив гравець
// і скільки ходів він зіграв у цих змінних
int hits = 0; //кількість влучань
int numberOfTurns = 0;//кількість ходів

// Дозвольте гравцеві продовжувати рух,
//поки він не потрапить на всі чотири кораблі
while (hits < 4) {
  int row, column;

  cout << "Selecting coordinates\n";

  // Попросіть у гравця рядок
  cout << "Виберіть номер рядка від 0 до 3: ";
  cin >> row;

  // Попросіть у гравця стовпець
  cout << "Виберіть номер стовпця від 0 до 3: ";
  cin >> column;

  // Перевірте, чи існує корабель у цих координатах
  if (ships[row][column]) {
    // Якщо гравець потрапив у корабель,
//видаліть його, встановивши значення на нуль
    ships[row][column] = 0;

    // Збільшити лічильник ударів
    hits++;

    // Скажіть гравцеві, що він влучив у корабель
// і скільки кораблів залишилося
```

```
    cout << "Hit! " << (4-hits) << " left.\n\n";
  } else {
    // Скажіть гравцеві, що він не попав
    cout << "Miss\n\n";
  }

  // Порахуйте, скільки ходів зробив гравець
  numberOfTurns++;
}
cout << "Перемога!\n";
cout << "Ви виграли в " << numberOfTurns << " ходів";
```

Приклад протоколу виконання цієї програми наведено на рисунку 5.9.

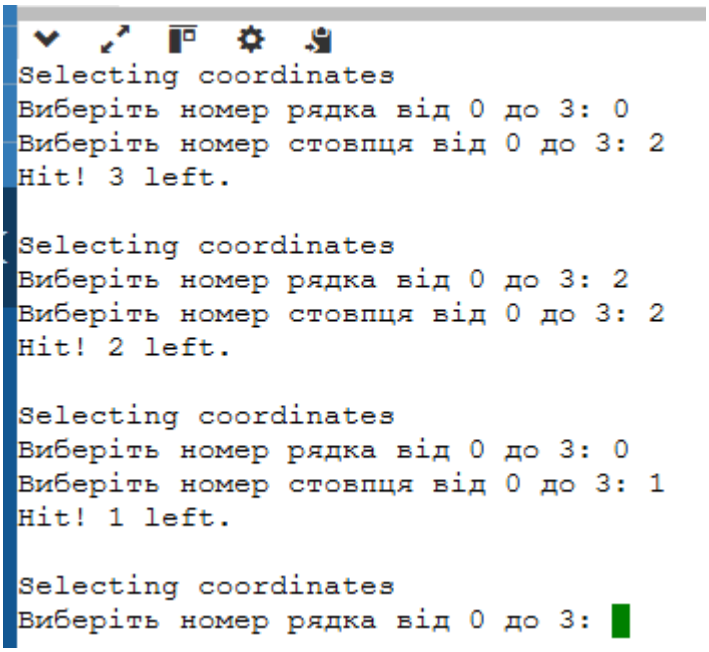


Рисунок 5.9 – Протокол виконання програми «Морський бій»

Контрольні питання

1. Що таке масив у мові програмування С і для чого він використовується?
2. Які переваги використання масивів порівняно з окремими змінними?
3. Як оголошується одновимірний масив у мові С? Наведіть приклад.
4. Як здійснюється доступ до окремого елемента масиву?
5. Що таке індекс масиву і з якого значення він починається в мові С?
6. Як можна ініціалізувати масив під час його оголошення?
7. Як обчислити суму або середнє значення елементів одновимірного масиву?
8. Як можна знайти максимальний або мінімальний елемент у масиві?
9. Що таке двовимірний масив і як його оголосити? Наведіть приклад.

10. Як звернутися до окремого елемента двовимірного масиву?
11. Як можна ініціалізувати двовимірний масив під час оголошення?
12. У чому різниця між одновимірним і багатовимірним масивами?
13. Як можна використати вкладені цикли для обробки елементів двовимірного масиву?
14. Наведіть приклади практичних задач, які доцільно розв’язувати з використанням масивів.

ТЕМА 6. Функції

6.1 Поняття про структурне програмування

На початку розвитку програмування (1950–60-ті роки) програми часто створювалися у вигляді послідовностей команд з безліччю операторів переходу (`goto`).

Такі програми називалися неструктурованими. З часом вони ставали трудними для розуміння, тестування й супроводу, особливо при збільшенні розміру проєктів.

У 1966 році Едсгер Дейкстра запропонував концепцію структурного програмування, що значно спростила процес розроблення та налагодження програм.

6.1.1 Теорема Бьома-Якопіні

Теорема (Бьома-Якопіні): виконуваний алгоритм може бути втілено з використанням лише трьох конкретних керівних структур: послідовного виконання, розгалужень, повторень (циклів).

Цю теорему було сформульовано та доведено італійськими математиками Коррадо Бьомом і Джузеппе Якопіні в їхній статті у 1966 р. У статті було описано методи перетворення неструктурованих алгоритмів на структуровані на прикладі створеної Бьомом мови програмування `P''`.

Структурна теорема Бьома-Якопіні була початком структурного програмування – парадигми програмування, яка виключає команди безумовного переходу (`goto`) й використовує виключно підпрограми, послідовне виконання, розгалуження (вибір) і цикли (ітерації).

Ця теорема є науковим положенням, використаним Е. Дейкстрою для обґрунтування його ідеї про використання в програмах лише трьох керуючих конструкцій: послідовного виконання, розгалужень і циклів.

Теорема Бьома-Якопіні не розв’язує питання про те, чи слід застосовувати структурне програмування для розробки програмного забезпечення. Деякі науковці використовували пуристській підхід до результату Бьома-Якопіні й

стверджували, що навіть такі інструкції, як `break` і `return` в середини циклів, є поганим підходом до розробки алгоритмів і всі цикли повинні мати єдину точка виходу. Пряме застосування теореми БьомаЯкопіні може привести до додавання додаткових локальних змінних до структурованої програми, а також до деякого дублювання коду. Останнє питання в цьому контексті називають «проблемою циклу з половиною». 1973 року С. Рао Косараджу довів, що можливо уникати додавання додаткових змінних в структурне програмування, якщо допускаються багаторівневі виходи довільної глибини з циклів.

Структурне програмування – це методологія розроблення програм, заснована на принципі поділу програми на логічно завершені частини (блоки) та використанні обмеженої кількості типів структур управління.

Мета структурного програмування – зробити програму зрозумілою, надійною, легкою для перевірки та модифікації.

Основна ідея – «розділяй і володарюй»: складна задача розбивається на простіші підзадачі.

6.1.2 Основні принципи структурного програмування

Структурне програмування – це основа сучасних методів розробки програмного забезпечення.

Воно забезпечує чіткість, передбачуваність та надійність програм.

Усі сучасні парадигми (у тому числі об’єктно-орієнтоване програмування) базуються на принципах структурного підходу.

Основні принципи структурного програмування:

1) *Декомпозиція* (поділ програми на модулі) – програма складається з підпрограм, функцій або процедур, кожна з яких виконує одну логічну дію.

2) Використання трьох базових структур управління:

– послідовність (sequence) – команди виконуються одна за одною;

– вибір (selection) – виконання залежить від умови (`if, switch`);

– повторення (iteration) – багаторазове виконання блоку (`for, while, do-while`).

Усі інші конструкції можуть бути виражені комбінацією цих трьох.

3) Відмова від безумовних переходів (`goto`*).*

Це забезпечує логічну прозорість і передбачуваність виконання.

4) Підхід зверху-вниз (top-down design)

Розроблення програми починається з головної задачі, яка поступово розкладається на менші підзадачі.

5) Незалежність модулів

Кожна функція може бути розроблена, протестована та налагоджена окремо.

6.1.3 Переваги та недоліки структурного програмування

Переваги структурного програмування:

– прозорість – програма легко читається;

– легке тестування й налагодження;

– зменшення кількості помилок;

– можливість колективної роботи над проектом;

– простота внесення змін і розширення.

Недоліки структурного підходу:

– обмеженість у роботі з дуже великими проектами, де потрібно керувати станом багатьох об’єктів (це згодом привело до появи об’єктно-орієнтованого програмування);

– моделі даних не завжди чітко пов’язані з функціями, які їх обробляють.

6.2 Поділ громіздкого алгоритму на частини

Під час розв’язування складних задач часто утворюється великий (громіздкий) алгоритм, який важко зрозуміти, налагодити або змінювати.

Щоб спростити розроблення, застосовується принцип *декомпозиції* – поділ складної програми на менші, логічно завершені частини.

Цей підхід є основою структурного програмування та сучасних методологій розроблення програмного забезпечення.

Декомпозиція алгоритму – це процес поділу великої задачі або програми на окремі підзадачі (модулі, блоки, підпрограми), кожна з яких виконує одну конкретну функцію.

Програма при цьому стає структурованою і багаторівневою:

– верхній рівень – головна програма (основна логіка);

– нижчі рівні – підпрограми, функції або процедури, які реалізують конкретні дії.

Мета поділу алгоритму:

– зменшення складності розробки;

– полегшення налагодження та тестування;

– можливість повторного використання частин програми;

– підвищення зрозумілості та наочності алгоритму;

– можливість командної розробки – різні модулі можуть створювати різні виконавці.

Основні принципи поділу:

– кожен модуль виконує одну логічну дію: «одна функція → одне призначення»;

– мінімальні зв’язки між модулями: модулі мають взаємодіяти через аргументи, не використовуючи спільні змінні;

– максимальна незалежність: зміни в одному модулі не повинні порушувати роботу інших;

– ієрархічна структура: головний алгоритм керує підпрограмами, які, у свою чергу, можуть викликати інші.

Методи поділу алгоритму:

1) за функціональною ознакою – алгоритм поділяється на частини відповідно до послідовності виконання дій, наприклад:

– введення даних;

– оброблення інформації;

– виведення результату.

2) за логічною структурою – виділяються окремі блоки для різних умов або повторюваних фрагментів.

3) за типами операцій. Наприклад, окремі функції для математичних обчислень, логічних перевірок, обміну даними тощо.

Порівняймо два варіанти коду, які виконують один і той самий функціонал (лістинг 6.1 та лістинг 6.2).

Лістинг 6.1 – *Громіздкий варіант (без поділу)*

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

int main() {
    int a, b, c;

    cout << "Введіть два числа: ";
    cin >> a >> b;
    c = a * a + b * b;
    if (c > 100)
        cout << "Сума квадратів більша за 100";
    else
        cout << "Сума квадратів не більша за 100";
    return 0;
     }
```

Кінець лістингу 6.1

Лістинг 6.2 – Покращений варіант (з поділом на частини)

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

int input() {
    int x;
    cout << "Введіть число: ";
    cin >> x;
    return x;
}

int sum_of_squares(int a, int b) {
    return a * a + b * b;
}
```

```
void output(int result) {
    if (result > 100)
        cout << "Сума квадратів більша за 100";
    else
        cout << "Сума квадратів не більша за 100";
}

int main() {
    int a = input();
    int b = input();
    int c = sum_of_squares(a, b);
    output(c);
    return 0;
     }
```

Кінець лістингу 6.2

Переваги такого поділу:

– код став зрозумілішим (підвищення читабельності коду);

– кожна функція виконує одну конкретну дію (спрощення структури програми);

– легко перевіряти і змінювати частини окремо (зручність у розробленні, тестуванні та відлагодженні, полегшення колективної роботи);

– можливість багаторазового використання підпрограм;

– легке розширення функціональності програми.

Практичні рекомендації:

1. Починайте розробку *з головної ідеї* програми.

2. Виділіть *основні функціональні частини* (етапи алгоритму).

3. Реалізуйте кожну частину у вигляді *окремої функції*.

4. Здійснюйте *поетапне налагодження* (спочатку прості частини, потім складніші).

5. Використовуйте *чіткі імена функцій* (`readData()`, `calculateSum()`, `printResult()`).

6.3 Стандартні функції і їх бібліотеки

6.3.1 Поняття функції

Функція – це *іменований блок коду*, який виконує певну дію та може повертати результат.

Функції допомагають *структурувати програму, зменшити повторення коду та полегшити налагодження*.

У мові C++ програма завжди починається з виконання *головної функції*:

```
int main() {
    // тіло програми
}
```

Існують два основні типи функцій:

1. *Стандартні функції* – уже визначені в бібліотеках C++.
2. *Функції користувача* – створені програмістом для конкретних завдань.

6.3.2 Стандартні функції мови C++

Стандартні функції входять до складу *бібліотек стандартної мови C++*.

Щоб скористатися ними, необхідно підключити відповідну бібліотеку (заголовковий файл) за допомогою директиви `#include`. Деякі бібліотеки стандартних функцій наведено в таблиці 6.1.

Таблиця 6.1 – Основні бібліотеки стандартних функцій

Бібліотека	**Призначення**
`<iostream>`	введення та виведення даних (`cin, cout`)
`<cmath>` або `<math.h>`	математичні функції (`sqrt, pow, sin, cos` тощо)
`<cstdlib>`	функції роботи з пам’яттю, генерація випадкових чисел, перетворення типів
`<ctime>`	робота з часом і датами
`<string>`	робота з рядками
`<iomanip>`	форматування виведення
`<fstream>`	робота з файлами
`<algorithm>`	стандартні алгоритми (сортування, пошук тощо)

6.3.3. Приклади використання стандартних функцій

Приклади використання стандартних функцій наведено в лістингах 6.3-6.5.

Лістинг 6.3 – Математичні функції (бібліотека <cmath>)

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

int main() {
    double x = 9.0;
    cout << "sqrt(x) = " << sqrt(x) << endl;    // корінь
// піднесення до степеня
    cout << "pow(2,3) = " << pow(2, 3) << endl;
    cout << "sin(0) = " << sin(0) << endl;      // синус
    cout << "abs(-5) = " << fabs(-5) << endl;   // модуль
     }
```

Кінець лістингу 6.3

Лістинг 6.4 – Функції роботи з рядками (<string> і <cstring>)

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

int main() {
    char str1[20] = "Hello";
    char str2[20] = "World";

    strcat(str1, str2);           // об’єднання
    cout << str1 << endl;         // HelloWorld
    cout << strlen(str1) << endl; // довжина рядка
     }
```

Кінець лістингу 6.4

Лістинг 6.5 – Функції генерації випадкових чисел (<cstdlib>, <ctime>)

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

int main() {
    srand(time(0));   // ініціалізація генератора
    int r = rand() % 100;  // випадкове число від 0 до 99
    cout << "Випадкове число: " << r;
     }
```

Кінець лістингу 6.5

6.4 Оголошення та виклик функцій користувача

Функції користувача – це функції, які визначає програміст самостійно.

Загальна форма оголошення функції:

```
тип_повернення ім’я_функції(список_параметрів) {
    // тіло функції
    return значення;
}
```

Розглянемо створення та використання функції користувача на прикладі лістингу 6.6.

Лістинг 6.6 – Приклад створення та використання функції користувача

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

int sum(int a, int b) {   // визначення функції
    return a + b;
}

int main() {
    int x = 5, y = 7;
    cout << "Сума = " << sum(x, y);   // виклик функції
      }
```

Кінець лістингу 6.6

6.5 Параметри функції

Функції можуть приймати аргументи – значення, які передаються під час виклику.

При передачі параметрів за значенням створюється копія змінної (лістинг 6.7), а при передачі за посиланням (лістинг 6.8) функція змінює реальне значення.

Лістинг 6.7 – Передача параметрів за значенням

```
void square(int n) {
    n = n * n;
```

```
    cout << "Всередині функції: " << n << endl;
}

int main() {
    int x = 4;
    square(x);
    cout << "Після виклику: " << x;  // x не змінюється
      }
```

Кінець лістингу 6.7

Лістинг 6.8 – Передача параметрів за посиланням

```
void squareRef(int &n) {
    n = n * n;
}

int main() {
    int x = 4;
    squareRef(x);
    cout << "Після виклику: " << x;  // x змінюється
      }
```

Кінець лістингу 6.8

6.6 Прототип функції

Якщо функція викликається раніше, ніж оголошена, потрібно спочатку вказати її прототип:

```
int sum(int, int);   // прототип

int main() {
    cout << sum(2, 3);
}

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

Прототип повідомляє компілятору про *тип функції і параметри*, але не містить її тіла.

6.7 Область видимості. Локальні і глобальні змінні

Область видимості (scope) – це частина програми, у межах якої можна звертатися до змінної або функції за її іменем.

Інакше кажучи, це – «зона дії» імені змінної.

У мові C++ ім’я змінної, функції або об’єкта доступне тільки в тій частині програми, де воно було оголошене. Види областей видимості наведено в таблиці 6.2.

Таблиця 6.2 – Типи областей видимості

Тип області	**Де оголошується**	**Де доступна**	**Тривалість існування**
Глобальна	поза будь-якими функціями	у всій програмі	протягом роботи програми
Локальна	всередині функції або блоку {}	лише в межах цього блоку	до завершення виконання блоку
Блочна (вкладена)	усередині іншого блоку {}	тільки в цьому внутрішньому блоці	до виходу з блоку
Параметри функції	у заголовку функції	лише всередині цієї функції	поки виконується функція

6.7.1 Глобальні змінні

Глобальні змінні оголошуються поза всіма функціями, зазвичай перед `main()`. Вони існують протягом усього виконання програми і доступні для всіх функцій (лістинг 6.9).

Лістинг 6.9 – Приклад використання глобальної змінної

```
#include <iostream>
using namespace std;
int g = 10;  // глобальна змінна
void show() {
    cout << "Глобальна змінна: " << g << endl;
}

int main() {
    cout << "У main(): " << g << endl;
    show();
    g = 25;   // зміна глобальної змінної
    show();
     }
```

Кінець лістингу 6.9

Перевагою використання глобальних змінних є зручність зберігання спільних даних для кількох функцій.

Недоліки:

– можуть викликати конфлікти імен;

– важко відстежувати зміни;

– порушують принцип модульності.

Тому рекомендується мінімізувати використання глобальних змінних, щоб уникнути помилок.

6.7.2 Локальні змінні

Локальні змінні оголошуються *всередині функцій або блоків* і доступні лише там. Вони створюються при вході в блок і знищуються при виході з нього (лістинг 6.10).

Лістинг 6.10 – Приклад використання локальної змінної

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

void test() {
    int a = 5;  // локальна змінна
    cout << "У функції test(): " << a << endl;
}

int main() {
    int a = 10;  // інша змінна з тим самим ім’ям
    cout << "У main(): " << a << endl;
    test();
      }
```

Кінець лістингу 6.10

У цьому прикладі змінні `a` у функціях *незалежні одна від одної*, навіть якщо мають однакове ім’я.

Аргументи (параметри) функції також є *локальними* змінними, які створюються під час виклику функції.

```
int sum(int a, int b) {
    return a + b;  // a і b — локальні для цієї функції
}
```

Після завершення виконання функції `a i b` знищуються.

6.7.3 Блокові змінні

У мові C++ можна оголошувати змінні *всередині окремих блоків коду*, обмежених фігурними дужками `{}`:

```
int main() {
    int x = 5;
    {
        int y = 10;
        cout << "x = " << x << ", y = " << y << endl;
    }
    // y тут вже не існує
    cout << "x = " << x << endl;
}
```

Змінна `y` існує лише всередині внутрішнього блоку. Змінна `x` доступна в усьому тілі функції `main().`

6.7.4 Тіньове перекриття імен (shadowing)

Якщо локальна змінна має те саме ім’я, що й глобальна, то у межах блоку використовується саме локальна змінна (лістинг 6.11).

Лістинг 6.11 – Приклад перекриття змінної

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

int x = 100;  // глобальна

int main() {
    int x = 5;   // локальна з тим самим ім’ям
    cout << x;   // виведе 5, а не 100
      }
```

Кінець лістингу 6.11

Для доступу до глобальної змінної, коли є така сама локальна, використовують *оператор області видимості* `::`:

```
cout << ::x;   // звернення до глобальної змінної
```

6.7.5 Статичні локальні змінні

Статичні змінні **(static)** зберігають своє значення між викликами функції (лістинг 6.12).

Лістинг 6.12 – Приклад використання статичної змінної

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

void counter() {
    static int count = 0;  // створюється один раз
    count++;
    cout << "Виклик № " << count << endl;
}

int main() {
    counter();
    counter();
    counter();
      }
```

Кінець лістингу 6.12

Результат виконання програми:

```
Виклик №1
Виклик №2
Виклик №3
```

6.7.6 Аргументи за замовчуванням

Аргументи за замовчуванням (default arguments) – це такі параметри функції, яким присвоюється типове значення, якщо під час виклику функції для них не подано значення.

Інакше кажучи, це значення параметрів, які функція отримує автоматично, якщо користувач їх не задає.

Типове значення (лістинг 6.13) задається у *заголовку функції* (у прототипі або визначенні):

```
тип_повернення ім’я_функції(тип параметр = значення, ...);
```

Лістинг 6.13 – Приклад використання аргументу за замовчуванням

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

void greet(string name = "Студент") {
    cout << "Привіт, " << name << "!" << endl;
}

int main() {
    greet("Світлана");   // Передаємо аргумент
    greet(); // Використовується значення за замовчуванням
     }
```

Кінець лістингу 6.13

Результат виконання програми:

```
Привіт, Світлана!
Привіт, Студент!
```

Правила використання аргументів за замовчуванням:

1) аргументи за замовчуванням *вказуються справа наліво*, тобто після параметрів без типових значень не можна ставити параметри з типовими;

Неправильно:

```
void f(int a = 5, int b); // помилка
```

Правильно:

```
void f(int a, int b = 5);
```

2) якщо типовий параметр задано у прототипі функції, то в оголошенні (тілі) його не потрібно повторювати (лістинг 6.15);

```
void print(char c = '*'); // прототип

void print(char c) {      // визначення
    for (int i = 0; i < 5; i++) cout << c;
    cout << endl;
}
```

3) аргументи за замовчуванням можна використовувати як у вбудованих типах (`int, double, string`), так і у користувацьких типах (структури, класи);

4) якщо функція викликається з меншою кількістю аргументів, то відсутні параметри заповнюються їхніми типовими значеннями (лістинг 6.14).

Лістинг 6.14 – Приклад з трьома параметрами

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

void showSum(int a, int b = 10, int c = 20) {
    cout << "Сума: " << a + b + c << endl;
}

int main() {
    showSum(5);        // 5 + 10 + 20 = 35
    showSum(5, 7);     // 5 + 7 + 20 = 32
    showSum(5, 7, 2);  // 5 + 7 + 2 = 14
      }
```

Кінець лістингу 6.14

Лістинг 6.15 – Аргументи за замовчуванням у прототипі

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

void volume(double a = 1.0, double b = 1.0, double c = 1.0);

int main() {
    volume(3.0, 4.0, 5.0);   // усі параметри задано
    volume(3.0, 4.0);        // третій — за замовчуванням
    volume(3.0); // другий і третій — за замовчуванням
    volume();    // усі — за замовчуванням
}

void volume(double a, double b, double c) {
    cout << "Об’єм = " << a * b * c << endl;
      }
```

Кінець лістингу 6.15

Переваги використання аргументів за замовчуванням:

– скорочення кількості перевантажених функцій – замість кількох варіантів із різною кількістю параметрів можна створити одну універсальну;

– зручність виклику функцій – можна задавати лише ті параметри, які справді потрібно змінити;

– підвищення читабельності та гнучкості коду.

Застереження: якщо значення за замовчуванням змінити лише в оголошенні, а не в прототипі, може виникнути невідповідність. Тому типові значення краще задавати в одному місці (зазвичай у прототипі).

Приклад програми з різними областями видимості наведено в лістингу 6.16.

Лістинг 6.16 – Змінні з різними областями видимості

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

int g = 50;  // глобальна змінна

void demo(int x) {
    int y = 10;     // локальна
    static int s = 0; // статична
    s++;
    cout << "x = " << x << ", y = " << y << ", g = " << g <<
", s = " << s << endl;
}

int main() {
    int a = 5;
    demo(a);
    demo(a);
      }
```

Кінець лістингу 6.16

6.8 Простори імен (namespace)

Простір імен (namespace) – це механізм у C++, який дозволяє групувати імена (змінні, функції, класи, константи) в окремі логічні області, щоб уникнути конфліктів імен у великих програмах.

У великих проектах можуть існувати функції або змінні з однаковими назвами. Простори імен дозволяють відокремити їх, щоб компілятор розрізняв, яке саме ім’я мається на увазі.

Синтаксис оголошення простору імен:

```
namespace ім’я_простору {
    // оголошення змінних, функцій, класів тощо
}
```

Приклад наведено в лістингу 6.17.

Лістинг 6.17 – Приклад використання простору імен

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

namespace Math {
    int sum(int a, int b) {
        return a + b;
    }
}

int main() {
// використання функції з простору Math
    cout << Math::sum(5, 7);
     }
```

Кінець лістингу 6.17

В лістингу 6.17 `Math::sum` означає, що функція `sum` із простору імен `Math`.

Уявімо, що в різних бібліотеках є функції з однаковою назвою:

```
int max(int a, int b);
double max(double a, double b);
```

А якщо підключити дві сторонні бібліотеки, у яких обидві мають функцію однакові імена (наприклад, `init()`), то компілятор не знатиме, яку саме викликати.

Вирішення: розмістити їх у різних просторах імен:

```
namespace Graphics {
```

```
    void init();
}

namespace Sound {
    void init();
}
```

Звертання до елементів простору імен можливе двома способами:

– через оператор області видимості `::`

```
Graphics::init();
Sound::init();
```

– за допомогою директиви `using namespace`

```
using namespace Graphics;

init();   // тепер можна викликати без Graphics::
```

Але краще не використовувати `using namespace` у великих програмах, бо це може знову призвести до конфліктів імен.

У C++ можна створювати простори імен всередині інших:

```
namespace Company {
    namespace IT {
        void info() {
            std::cout << "IT Department" << std::endl;
        }
    }
}

int main() {
    Company::IT::info();
}
```

Всі елементи стандартної бібліотеки C++ (наприклад `cout`, `cin`, `string`, `vector`) належать до простору імен `std`.

```
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
}
```

Щоб не писати `std::` кожного разу, часто використовують:

```
using namespace std;
```

або лише окремі елементи:

```
using std::cout;
using std::endl;
```

Приклад використання різних просторів імен наведено в лістингу 6.18.

Лістинг 6.18 – Приклад використання просторів імен

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

namespace Math {
    int add(int a, int b) { return a + b; }
    int sub(int a, int b) { return a - b; }
}

namespace Logic {
    bool equal(int a, int b) { return a == b; }
}

int main() {
    cout << "5 + 3 = " << Math::add(5, 3) << endl;
    cout << "5 - 3 = " << Math::sub(5, 3) << endl;
    cout << "5 == 3 ? " << Logic::equal(5, 3) << endl;
      }
```

Кінець лістингу 6.18

Переваги використання просторів імен:

– уникнення конфліктів імен – різні бібліотеки можуть містити функції з однаковими назвами;

– логічна організація коду – можна групувати пов’язані елементи;

– зручність у великих проєктах – легко підтримувати, розширювати та інтегрувати модулі.

Недоліки і застереження:

– надмірне використання `using namespace` може призвести до зіткнення імен;

– якщо простори імен дуже вкладені, це може ускладнити читання коду;

– не рекомендується писати `using namespace std;` у заголовкових файлах (з розширенням `.h`).

6.9 Перевантаження та шаблони функцій

6.9.1 Перевантаження функцій

Перевантаження функцій – це можливість створювати кілька функцій з однаковим іменем, але з різними списками параметрів (типами або кількістю).

Компілятор автоматично визначає, яку саме функцію викликати, залежно від переданих аргументів.

Синтаксис:

```
тип_повернення ім’я_функції(список_параметрів);
```

Дозволено мати кілька таких оголошень з різними параметрами (лістинг 6.19).

Лістинг 6.19 – Приклад перевантаження функцій

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

int add(int a, int b) {
    return a + b;
}

double add(double a, double b) {
    return a + b;
}

string add(string a, string b) {
    return a + " " + b;
}

int main() {
    cout << add(3, 4) << endl; // виклик add(int, int)
    cout << add(2.5, 4.1) << endl; /* виклик add(double,
double)*/
    cout << add("Hello", "World") << endl; /* виклик
add(string, string)*/
     }
```

Кінець лістингу 6.19

Компілятор виконує механізм підбору перевантаженої функції – *overload resolution*:

– шукає усі функції з таким самим ім’ям;

– обирає ту, параметри якої *найкраще відповідають* переданим аргументам;

– якщо знайти однозначно не вдалося – *помилка компіляції*.

Правила перевантаження:

– можна перевантажувати функції, які мають однакове ім’я, але різні типи або кількість параметрів;

– не можна перевантажувати функції, які відрізняються лише типом, який повертається або порядком однакових параметрів.

Неправильно:

```
int f(int x);
double f(int x); /* помилка – відрізняється лише типом повернення*/
```

Правильно показано в лістингу 6.20.

Лістинг 6.20 – Приклад перевантаження функцій з різною кількістю параметрів

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

void show(int a) {
    cout << "Один параметр: " << a << endl;
}

void show(int a, int b) {
    cout << "Два параметри: " << a << ", " << b << endl;
}

int main() {
    show(5);
    show(5, 10);
      }
```

Кінець лістингу 6.20

Якщо використовуються *аргументи за замовчуванням*, перевантаження треба застосовувати обережно, адже виклики можуть бути неоднозначними.

```
void print(int x = 0);
void print(int x, int y);

print(); // помилка: неясно, яку функцію викликати
```

Переваги перевантаження:

– зручніше запам’ятовувати функції (одне ім’я – кілька варіантів);

– підвищує читабельність і гнучкість коду;

– дозволяє працювати з різними типами даних однаковим способом.

6.9.2 Шаблони функцій

Шаблон функції (`template` function) – це універсальна функція, яка може працювати з будь-яким типом даних, визначеним під час виклику.

Шаблони – це основа узагальненого програмування (generic programming).

Синтаксис шаблону функції:

```
template <typename T>
тип_повернення ім’я_функції(параметри_типу T);
```

де `T` – формальний параметр типу, який замінюється на реальний тип під час виклику функції (лістинг 6.21).

Лістинг 6.21 – Приклад простого шаблону

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

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add(3, 4) << endl;       // T = int
    cout << add(2.5, 4.1) << endl;   // T = double
    cout << add(string("Hi"), string(" there")) << endl; // T
= string
     }
```

Кінець лістингу 6.21

В лістингу 6.21 використано одну функцію – для всіх типів, які підтримують оператор +. Можна також використовувати кілька параметрів типу (лістинг 6.22).

Лістинг 6.22 – Кілька параметрів типу

```
template <typename T1, typename T2>
void show(T1 a, T2 b) {
    cout << a << " і " << b << endl;
}

int main() {
    show(5, "текст");
    show(3.14, 42);
     }
```

Кінець лістингу 6.22

Тип шаблону можна:

– *визначати неявно* – компілятор сам розпізнає типи;

– *вказати явно* при виклику:

```
add<int>(3, 5);
add<double>(2.5, 4.2);
```

Слід пам’ятати, що шаблон працює лише з тими типами, які підтримують потрібні операції. Наприклад, якщо функція використовує оператор +, то тип T повинен мати його реалізацію.

C++ дозволяє одночасно використовувати і перевантажені, і шаблонні функції (лістинг 6.23).

Лістинг 6.23 – Поєднання шаблонів функцій та перевантаження

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

void show(int a) { cout << "Ціле: " << a << endl; }

template <typename T>
void show(T a) { cout << "Шаблон: " << a << endl; }

int main() {
    show(5);      // викликає звичайну функцію
```

```
    show(3.14);   // викликає шаблон
    }
```

Кінець лістингу 6.23

Якщо є і звичайна, і шаблонна функція, компілятор *спершу вибирає звичайну* (якщо тип точно збігається).

Переваги шаблонів:

– один код для будь-яких типів даних;

– економія часу розробки;

– універсальність і повторне використання.

Недоліки шаблонів:

– помилки при компіляції можуть бути складними для розуміння;

– код шаблонів може збільшувати розмір програми (через створення копій для різних типів);

– шаблони ускладнюють налагодження.

6.10 Рекурсія

Рекурсія – це процес, при якому *функція викликає саму себе*. Функція, яка викликає сама себе, називається рекурсивною.

Ідея рекурсії полягає в тому, що велика задача розв'язується шляхом поділу на менші *підзадачі того самого типу*.

Наприклад, 4! – це:

```
factorial(4)
= 4 * factorial(3)
= 4 * (3 * factorial(2))
= 4 * (3 * (2 * factorial(1)))
= 4 * (3 * (2 * 1))
= 24
```

Рекурсивна функція обов'язково повинна мати два основні компоненти:

1) *Базовий випадок (умова завершення)* – момент, коли рекурсія припиняється.

2) *Рекурсивний виклик* – виклик самої себе для обробки меншої підзадачі.

Загальний вигляд рекурсивної функції:

```
тип функція(параметри) {
    if (умова_завершення)
        return результат;
    else
        return    функція(нові_параметри);    /*    рекурсивний
виклик*/
}
```

Прості приклади рекурсії наведено в лістингах 6.24-6.25.

Лістинг 6.24 – Обчислення факторіала числа n!

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

int factorial(int n) {
    if (n <= 1) return 1;          // базовий випадок
    else return n * factorial(n - 1); /* рекурсивний виклик*/
}

int main() {
    cout << "5! = " << factorial(5);
     }
```

Кінець лістингу 6.24

Хід виконання:

```
factorial(5) → 5 * factorial(4) → 5 * 4 * factorial(3) → … → 5
* 4 * 3 * 2 * 1 = 120.
```

Послідовність Фібоначчі визначається як:

```
F(0) = 0, F(1) = 1,
F(n) = F(n-1) + F(n-2)
```

Лістинг 6.25 – Обчислення чисел Фібоначчі

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

int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
```

```
}

int main() {
    for (int i = 0; i < 10; i++)
        cout << fib(i) << " ";
     }
```

Кінець лістингу 6.25

Багато задач, які можна розв’язати рекурсивно, можна також розв’язати *ітераційно* (за допомогою циклів).

Проте рекурсія часто робить програму зрозумілішою, ближчою до математичного опису задачі. Порівняння цих двох механізмів наведено в таблиці 6.3.

Таблиця 6.3 – Порівняння рекурсії та ітерації

Ознака	**Рекурсія**	**Ітерація**
Принцип	Функція викликає саму себе	Повторення через цикл
Необхідна умова	Базовий випадок	Умова виходу з циклу
Використання пам’яті	Більше (створюється стек викликів)	Менше
Швидкодія	Повільніша	Швидша
Зручність для розуміння	Вища при математичних формулах	Краща для простих обчислень

Кожен рекурсивний виклик:

–створює нову копію локальних змінних функції;

–зберігає поточний стан у *стеку викликів*;

–після повернення – продовжує виконання з місця виклику.

Якщо базовий випадок не задано або він недосяжний – виникає *нескінченна рекурсія і переповнення стека (stack overflow)*.

Види рекурсії:

–пряма рекурсія – функція безпосередньо викликає саму себе:

```
void A() { A(); }  // приклад прямої рекурсії
```

–непряма (взаємна) рекурсія – функція викликає іншу функцію, яка потім знову викликає першу:

```
void A();
void B();
void A() { cout << "A "; B(); }
```

```
void B() { cout << "B "; A(); }

int main() { A(); } // нескінченна взаємна рекурсія
```

Приклади використання рекурсивних функцій наведено в лістингах 6.26 - 6.27.

Лістинг 6.26 – Найбільший спільний дільник (алгоритм Евкліда)

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

int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

int main() {
    cout << "НСД(24, 18) = " << gcd(24, 18);
     }
```

Кінець лістингу 6.26

Хід виконання:

```
gcd(24,18) → gcd(18,6) → gcd(6,0) → результат 6.
```

Лістинг 6.27 – Рекурсивний підрахунок цифр у числі

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

int digits(int n) {
    if (n < 10) return 1;
    return 1 + digits(n / 10);
}

int main() {
    cout << "Кількість цифр: " << digits(12345);
     }
```

Кінець лістингу 6.27

Рекурсивна функція може викликати себе кілька разів у межах одного виклику. Наприклад, функція Аккермана – класичний приклад сильної рекурсії:

```
int A(int m, int n) {
    if (m == 0) return n + 1;
    if (n == 0) return A(m - 1, 1);
    return A(m - 1, A(m, n - 1));
}
```

Переваги рекурсії:

–простий і наочний спосіб опису задач, що мають *самоподібну структуру*;

–коротший і зрозуміліший код;

–природне відображення математичних формул.

Недоліки рекурсії:

–велике споживання пам’яті (через стек викликів);

–нижча швидкодія порівняно з ітераційним підходом;

–ризик переповнення стека при глибокій рекурсії.

Рекомендації щодо використання:

–завжди визначати базовий випадок;

–для великих обсягів даних – краще ітераційне рішення;

–рекурсію доцільно застосовувати там, де задача природно визначається рекурсивно (наприклад, дерева, пошук, обходи графів).

Контрольні питання

1. У чому полягає суть структурного програмування?
2. Які основні принципи структурного програмування ви знаєте?
3. Чому доцільно ділити великі програми або алгоритми на окремі функціональні частини?
4. Що таке функція у мові програмування C/C++?
5. Яке призначення мають стандартні бібліотечні функції? Наведіть приклади таких бібліотек.
6. Як оголошується та викликається функція користувача у мові C/C++?
7. Яку роль відіграє прототип функції? Чому його необхідно вказувати перед `main()`?
8. Які види параметрів функцій ви знаєте і як вони передаються у функцію?
9. Що таке область видимості змінних?
10. У чому відмінність між локальними і глобальними змінними? Наведіть приклади.

11. Що таке аргументи за замовчуванням і як вони оголошуються у функціях мови C++?

12. Що таке простір імен (`namespace`) і для чого його використовують у програмах на C++?

13. Що означає поняття «перенавантаження функцій» у мові C++? Наведіть приклад.

14. Що таке шаблони функцій і яку перевагу вони дають програмісту?

15. Що таке рекурсія?

16. Наведіть приклад рекурсивної функції та поясніть принцип її роботи.

ТЕМА 7. Адреси даних та вказівники

7.1 Адреси даних

Усі дані, що використовуються програмою, зберігаються в *оперативній пам'яті комп'ютера*.

Кожна комірка пам'яті має *унікальну адресу* – ціле число, яке однозначно визначає її положення (рисунок 7.1).

```
int x = 8;
```

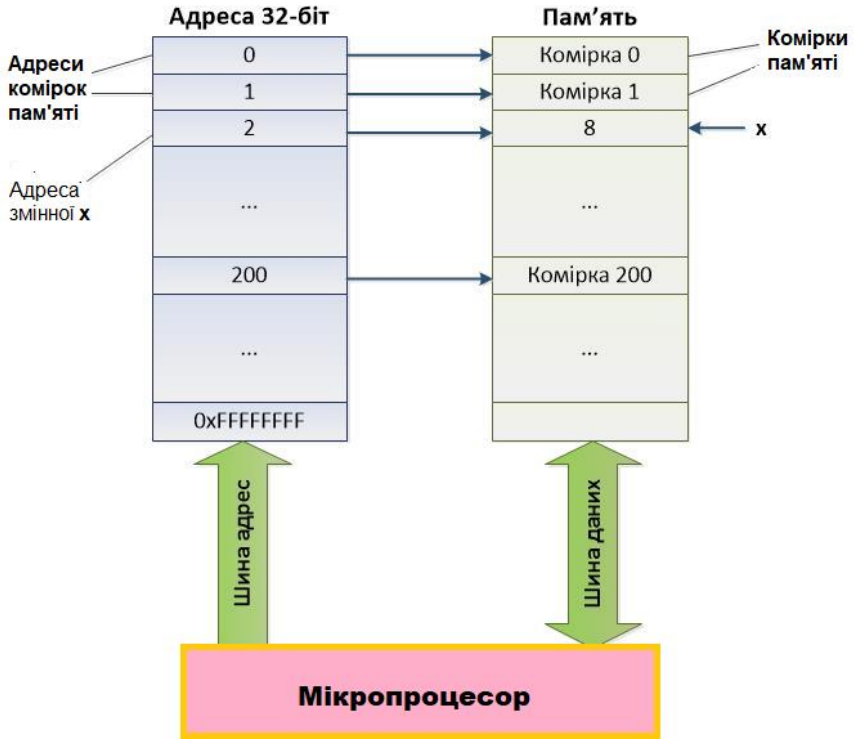


Рисунок 7.1 – Адреси в пам'яті

У мові C++ будь-яка змінна, окрім свого імені та значення, має ще й *адресу пам'яті*, де зберігається її значення.

7.1.1 Оператор & – отримання адреси

Для отримання адреси змінної використовується унарний оператор &.

```
int x = 8;
cout << &x;   // виведе адресу змінної x
```

Адреси зазвичай відображаються у *шістнадцятковій системі числення*, наприклад `0x7ffee82c0b2c.`

7.1.2 Розмір змінних

Розмір пам'яті, який займає змінна певного типу, можна дізнатися за допомогою оператора `sizeof()`:

```
cout << sizeof(int);   // 4
cout << sizeof(double); // 8
cout << sizeof(float); // 4
cout << sizeof(char); // 1
```

7.2 Вказівники

Вказівник (pointer) – це змінна, яка *зберігає адресу* іншої змінної, тобто вказівник це змінна, значенням якої є адреса комірки пам'яті. Тобто вказівник посилається на блок даних із області оперативної пам'яті, при чому в самий його початок. Вказівник може посилатися на змінну або функцію.

7.2.1 Оголошення вказівників

Вказівник оголошуємо за тим самим принципом що і змінна, тільки перед ім'ям змінної ставимо символ * (символ * означає, що змінна є вказівником):

```
тип_даних * ім'я_змінної;
```

Наприклад:

```
int *ptr_a; // вказівник;
int b = 8; // проста змінна
ptr_a = &b; // взяття адреси і присвоєння її вказівнику
```

В програмуванні прийнято додавати до імені вказівника префікс `ptr`, таким чином, не дозволяючи плутати прості змінні і змінні вказівників.

```
int *p;      // оголошення вказівника на int
double *q;   // оголошення вказівника на double
```

7.2.2 Присвоєння адреси

```
int x = 8;
int *p = &x;   // у p записується адреса змінної x
```

Ілюстрація присвоєння значення вказівникові наведена на рисунку 7.2.

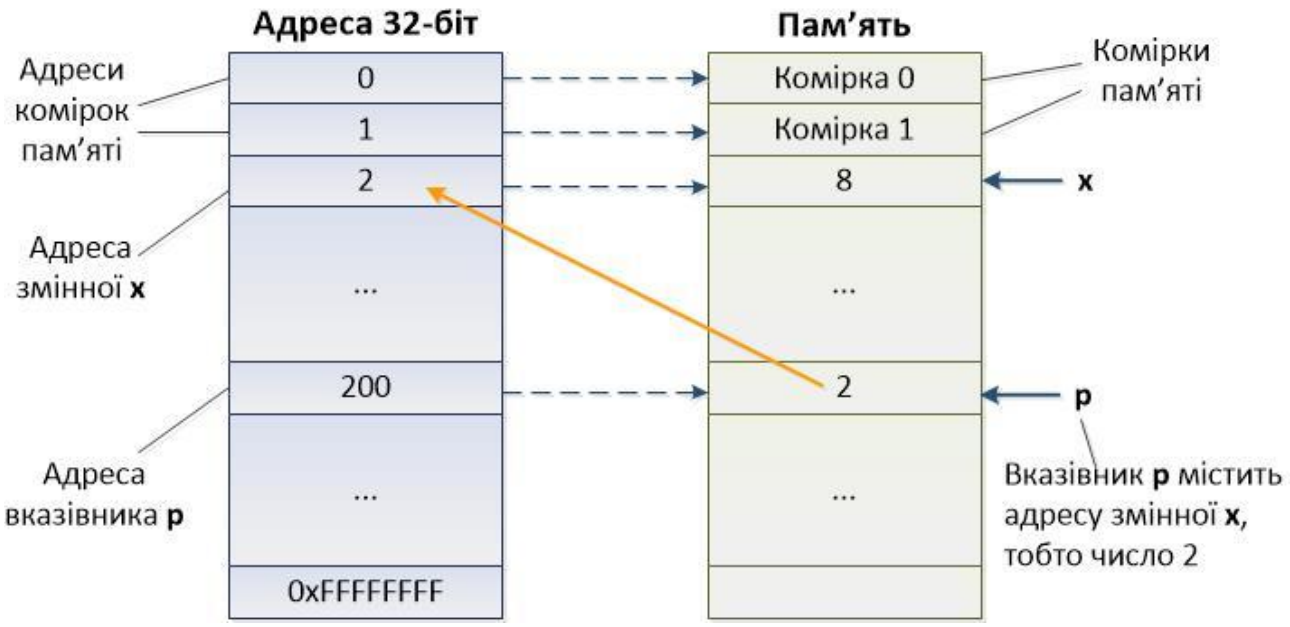


Рисунок 7.2 – Присвоєння адреси змінної вказівникові

7.2.3 Розіменування (*)

Щоб отримати значення, на яке вказує вказівник, використовують оператор розіменування * (рисунок 7.3):

```
cout << *p;   // виведе 8
```

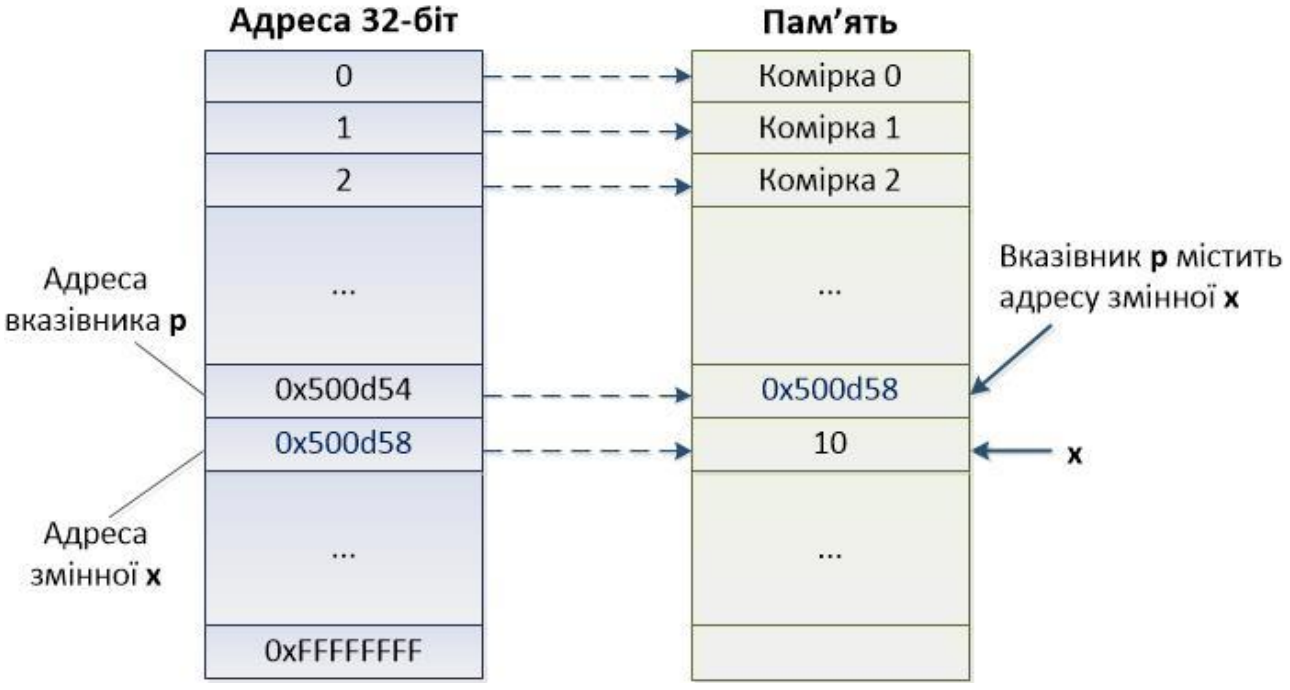


Рисунок 7.3 – Розіменування вказівника

Розіменування дозволяє *звертатися до значення змінної через адресу*.

Приклад

```
int x = 10;
int *p = &x;
*p = 20;// через вказівник змінюємо значення змінної x
cout << x;  // 20
```

7.2.4 Арифметика вказівників

Вказівники можна збільшувати або зменшувати.

При цьому вони переміщуються не на 1 байт, а на *розмір типу*, на який вони вказують.

```
int arr[3] = {10, 20, 30};
int *p = arr;
cout << *p << endl;     // 10
p++;
cout << *p << endl;     // 20
```

Арифметику вказівників застосовують *лише до масивів або динамічних блоків пам'яті*.

7.3 Динамічна пам'ять

Статичне виділення пам'яті – розмір масиву або змінних визначається під час компіляції.

Динамічне виділення пам'яті – пам'ять виділяється під час виконання програми.

У C++ для роботи з динамічною пам'яттю використовують оператори `new` та `delete`.

```
int *p = new int; //виділити пам'ять під одне ціле число
*p = 100;
delete p;             // звільнити пам'ять
Динамічний масив
int *arr = new int[5];
for (int i = 0; i < 5; i++)
    arr[i] = i * 10;
```

```
delete[] arr;    // звільнення пам’яті для масиву
```

Невиконання `delete` призводить до *витоків пам’яті (memory leak)*.

7.4 Вказівники на вказівники

Вказівники можуть посилатися на інші вказівники. При цьому в комірці пам’яті, на котру буде посилатися перший вказівник, буде міститися не значення, а адреса іншого вказівника.

```
int a = 10;
int *p = &a;
int **pp = &p;
cout << **pp;   // 10
```

Кількість символів * при оголошенні вказівника показує порядок вказівника. Щоб отримати доступ до значення, на яке посилається вказівник його слід розіменувати декілька разів.

```
int a = 7;
int *ptr_1 = &a;     // вказівник на адресу змінної a
int **ptr_2 = &ptr_1;// вказівник на адресу вказівника
// ptr_1 (ptr_1 вказує на змінну a)

cout << "*ptr_1  = " <<  *ptr_1  << endl;
// розіменування вказівника на вказівник  **
cout << "**ptr_2 = " << **ptr_2  << endl;
// так виведеться, лише адреса, на яку вказує
// вказівник "ptr_2" тобто адреса вказівника ptr_1
cout << "*ptr_2  = " << *ptr_2   << endl;
```

7.5 Звертання до даних через вказівники

7.5.1 Вказівники як аргументи функцій

Вказівники дозволяють функції *змінювати змінні, передані їй*:

```
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```

```
int main() {
    int x = 5, y = 7;
    swap(&x, &y);
    cout << x << " " << y;   // 7 5
}
```

7.5.2 Вказівники на структури

```
struct Point {
    int x, y;
};

Point A = {3, 4};
Point *p = &A;
// доступ до полів структури через ->
cout << p->x << ", " << p->y;
p->x еквівалентне (*p).x.
```

7.5.3 Вказівники та масиви

Ім’я масиву є вказівником на його перший елемент:

```
int arr[3] = {1, 2, 3};
cout << arr << endl;     // адреса першого елемента
cout << *arr << endl;    // 1
```

Тому можна використовувати вказівники для обходу масиву:

```
for (int *p = arr; p < arr + 3; p++)
    cout << *p << " ";
```

7.5.4 Вказівники та безпека

Неправильне використання вказівників може спричинити:

–звернення до неіснуючої пам’яті;

–витоки пам’яті;

–помилки типу “segmentation fault”.

Тому рекомендується:

–ініціалізувати вказівники (`int *p = nullptr;`);

–після `delete` – обнуляти (`p = nullptr;`);

–використовувати *розумні вказівники* (`unique_ptr, shared_ptr`).

Отже, *вказівники* – це один із найпотужніших, але й найнебезпечніших механізмів мови C++.

Вони дозволяють безпосередньо працювати з пам’яттю, передавати великі об’єкти у функції без копіювання, створювати динамічні структури даних.

Використання сучасних інструментів (`new, delete, smart pointers`) робить роботу з пам’яттю безпечнішою та ефективнішою.

Контрольні питання

1. Що таке адреса даних у пам’яті комп’ютера?
2. Що таке вказівник у мові програмування C/C++ і яке його призначення?
3. Як оголошується змінна-вказівник у мові C? Наведіть приклад.
4. Який оператор використовується для отримання адреси змінної?
5. Який оператор використовується для доступу до значення, на яке вказує вказівник?
6. Що відбувається при виконанні операції розіменування вказівника?
7. Як ініціалізувати вказівник і чому небезпечно залишати його неініціалізованим?
8. Що таке нульовий вказівник (NULL або nullptr) і для чого його використовують?
9. Як здійснюється арифметика вказівників і в яких випадках це може бути корисно?
10. Що таке динамічне виділення пам’яті?
11. Які функції стандартної бібліотеки мови C використовуються для роботи з динамічною пам’яттю?
12. Як у мові C++ виконуються операції виділення і звільнення пам’яті?
13. Що таке вказівник на вказівник і як його оголошують?
14. Як можна передати змінну у функцію через вказівник? Які переваги цього методу?
15. Наведіть приклад програми, у якій створюється динамічний масив за допомогою вказівників.

ТЕМА 8. Опрацювання масивів засобами мови C/C++

8.1 Звертання до елементів масиву через індекси та через вказівники

Масив – це послідовність елементів одного типу, розміщених у пам'яті підряд. Кожен елемент має:

– порядковий номер (індекс);

– адресу в пам'яті.

Ім'я масиву є *вказівником на його перший елемент*.

Традиційний спосіб доступу до елементів масиву – через індекс у квадратних дужках (лістинг 8.1).

Лістинг 8.1 – Звернення до елементів масиву через індекс

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

int main() {
    int arr[5] = {10, 20, 30, 40, 50};

    for (int i = 0; i < 5; i++)
        cout << "arr[" << i << "] = " << arr[i] << endl;

    return 0;
      }
```

Кінець лістингу 8.1

Оскільки елементи розміщені *послідовно*, то:

– адреса наступного елемента більша від попередньої на *розмір одного елемента*;

– вираз `&arr[i]` – адреса i-го елемента;

– ім'я масиву (`arr`) – це адреса *першого елемента*: `arr == &arr[0]`.

```
cout << "Адреса першого елемента: " << arr << endl;
cout << "Адреса другого елемента: " << &arr[1] << endl;
```

Оскільки ім’я масиву – це вказівник, ми можемо працювати з масивом *через арифметику вказівників* (лістинг 8.2).

Основна ідея:

```
arr[i]  ≡  *(arr + i)
```

Тобто:

– `arr + i` – адреса i-го елемента;

– `*(arr + i)` – значення i-го елемента.

Лістинг 8.2 – Звернення до елементів масиву через вказівник

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

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int *p = arr;   // вказівник на перший елемент масиву

    for (int i = 0; i < 5; i++)
        cout << "*(p + " << i << ") = " << *(p + i) << endl;

    return 0;
      }
```

Кінець лістингу 8.2

У C++ арифметичні операції з вказівниками враховують *розмір типу*, на який вони вказують.

Якщо:

```
int *p = arr;
```

тоді:

– `p + 1` – це не просто `+1, a + sizeof(int)` байт у пам’яті;

– `*(p + 1)` – другий елемент масиву;

– `*(p + i)` – і-й елемент.

Для будь-якого масиву `arr` можна скласти таблицю еквівалентності доступу (таблиця 8.1).

Таблиця 8.1 – Еквівентність доступу до елементів масиву

Операція	**Значення**
`arr[i]`	`*(arr + i)`
`&arr[i]`	`arr + i`
`*p`	поточний елемент
`*(p + i)`	елемент через відступ від початку масиву
`p[i]`	`*(p + i)`

Тобто синтаксис `p[i]` можна застосовувати не лише до масивів, а й до будь-якого вказівника на послідовність елементів.

Вказівник можна збільшувати або зменшувати для переходу між елементами (лістинг 8.3).

Лістинг 8.3 – Переміщення між елементами масиву за допомогою вказівника

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

int main() {
    int arr[4] = {5, 10, 15, 20};
    int *p = arr;

    cout << *p << endl;   // 5
    p++;                  // перехід до наступного елемента
    cout << *p << endl;   // 10
    p += 2;               // ще два елементи вперед
    cout << *p << endl;   // 20
      }
```

Кінець лістингу 8.3

Ми можемо обійти весь масив *без індексів*, лише через вказівник (лістинг 8.4).

Лістинг 8.4 – Використання вказівників у циклах

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

int main() {
    int arr[5] = {2, 4, 6, 8, 10};
    int *p = arr;
```

```
    while (p < arr + 5) {
        cout << *p << " ";
        p++;    // перехід до наступного елемента
    }
      }
```

Кінець лістингу 8.4

Можна змінювати значення елементів масиву, використовуючи вказівник (лістинг 8.5).

Лістинг 8.5 – Модифікація елементів через вказівник

```
int arr[3] = {1, 2, 3};
int *p = arr;

for (int i = 0; i < 3; i++)
    *(p + i) *= 2;

for (int i = 0; i < 3; i++)
    cout << arr[i] << " ";
```

Кінець лістингу 8.5

Вивід:

```
2 4 6
```

Для двовимірного масиву:

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

– `a` – вказівник на рядок;

– `*a` – вказівник на перший елемент рядка;

– `*(*(a + i) + j)` – доступ до елемента `[i][j]`.

```
cout << a[1][2];          // 6
cout << *(*(a + 1) + 2);  // 6
```

Вказівники особливо важливі при роботі з динамічною пам’яттю:

```
int n;
cin >> n;

int *arr = new int[n]; // створення динамічного масиву
```

```
for (int i = 0; i < n; i++)
    arr[i] = i * 10;

delete[] arr;           // звільнення пам’яті
```

Порівняння двох способів доступу до елементів масиву наведено в таблиці 8.2.

Таблиця 8.2 – Порівняння способів доступу до елементів масиву

Критерій	**Доступ через індекс**	**Доступ через вказівник**
Синтаксис	`arr[i]`	`*(arr + i)`
Зручність	Простий і зрозумілий	Вимагає знань про адреси
Гнучкість	Лише статичні масиви	Підтримує динамічну пам’ять
Продуктивність	Однакова	Однакова (після компіляції)

Компілятор C++ трактує обидва варіанти *однаково ефективно*, тому різниця лише у стилі та контексті використання.

Поради:

– для динамічних структур краще використовувати вказівники;

– для звичайних масивів індекси зрозуміліші;

– при роботі з вказівниками контролюйте межі масиву, щоб уникнути виходу за межі пам’яті.

8.2 Типові алгоритми опрацювання одновимірних масивів

Типові задачі при роботі з масивами – це *обробка елементів*, *пошук*, *впорядкування*, *фільтрація*, *агрегація* тощо.

Типові алгоритми опрацювання одновимірних масивів наведені в лістингах 8.6-8.12.

Лістинг 8.6 – Пошук максимального і мінімального елементів

```
int max = arr[0];
int min = arr[0];
```

```
for (int i = 1; i < N; i++) {
    if (arr[i] > max) max = arr[i];
    if (arr[i] < min) min = arr[i];
}

cout << "Максимум = " << max << endl;
cout << "Мінімум = " << min << endl;
```

Кінець лістингу 8.6

Лістинг 8.7 – Обчислення суми та середнього значення

```
int sum = 0;
for (int i = 0; i < N; i++)
    sum += arr[i];

double avg = (double)sum / N;

cout << "Сума = " << sum << endl;
cout << "Середнє = " << avg << endl;
```

Кінець лістингу 8.7

Лістинг 8.8 – Підрахунок кількості елементів, що задовольняють умову

```
// Підрахунок кількості додатних елементів
int count = 0;
for (int i = 0; i < N; i++)
    if (arr[i] > 0)
        count++;

cout << "Додатних елементів: " << count;
```

Кінець лістингу 8.8

Лістинг 8.9 – Пошук елемента (лінійний пошук)

```
int x;
cout << "Введіть елемент для пошуку: ";
cin >> x;

bool found = false;
for (int i = 0; i < N; i++)
    if (arr[i] == x) {
        found = true;
        cout << "Елемент знайдено на позиції " << i << endl;
```

```
        break;
    }

if (!found) cout << "Елемент не знайдено.";
```

Кінець лістингу 8.9

Лістинг 8.10 – Заміна елементів за умовою

```
// усі від’ємні елементи зробити нулями:
for (int i = 0; i < N; i++)
    if (arr[i] < 0)
        arr[i] = 0;
```

Кінець лістингу 8.10

Лістинг 8.11 – Обчислення кількості входжень певного елемента

```
int x, count = 0;
cin >> x;

for (int i = 0; i < N; i++)
    if (arr[i] == x)
        count++;

cout << "Елемент " << x << " зустрічається " << count << "
разів.";
```

Кінець лістингу 8.11

Лістинг 8.12 – Зсув елементів

```
//Зсув вліво на 1 елемент:
int temp = arr[0];
for (int i = 0; i < N - 1; i++)
    arr[i] = arr[i + 1];
arr[N - 1] = temp;
//Зсув вправо на 1 елемент:
int temp = arr[N - 1];
for (int i = N - 1; i > 0; i--)
    arr[i] = arr[i - 1];
arr[0] = temp;
```

Кінець лістингу 8.12

Практичне застосування типових алгоритмів опрацювання масивів:

– обробка вимірювань датчиків;

– аналіз статистичних даних;

– пошук у базах даних;

– робота з файлами з числовими або символьними даними;

– реалізація алгоритмів машинного навчання на базовому рівні.

8.3 Сортування масиву

8.3.1 Сортування методом обміну (бульбашкою)

Сортування методом обміну (бульбашкою) – це простий алгоритм, який послідовно порівнює та *міняє місцями сусідні елементи масиву*, якщо вони стоять у неправильному порядку, доки весь масив не буде відсортовано. Його назва походить від того, що найбільші (або найменші) елементи поступово «спливають» на свої кінцеві позиції, ніби бульбашки у воді.

Кроки алгоритму:

– порівняння сусідів – алгоритм проходить по масиву, порівнюючи кожен елемент із наступним;

– обмін – якщо елементи не відповідають умові сортування (наприклад, для сортування за зростанням, коли попередній елемент більший за наступний), вони міняються місцями;

– повторення – цей процес порівняння та обміну повторюється кілька разів. З кожним проходом найбільший елемент (при сортуванні за зростанням) «спливає» на кінець масиву, і в наступному проході вже не розглядається;

– завершення – алгоритм завершує роботу, коли цілий масив стає відсортованим, що визначається тим, що в одному з проходів не було зроблено жодного обміну.

Особливості алгоритму:

– алгоритм є одним із найпростіших для розуміння та реалізації;

– він не є ефективним для великих масивів, оскільки має квадратичну часову складність $O(n^2)$, що означає, що час виконання зростає дуже швидко зі збільшенням розміру масиву;

– здебільшого використовується в навчальних цілях для демонстрації базових принципів сортування (лістинг 8.13).

Лістинг 8.13 – Сортування методом обміну (бульбашкою)

```
for (int i = 0; i < N - 1; i++)
    for (int j = 0; j < N - i - 1; j++)
        if (arr[j] > arr[j + 1])
            swap(arr[j], arr[j + 1]);
```

Кінець лістингу 8.13

8.3.2 Сортування методом вибору

Сортування методом вибору (або селекційне сортування) – це простий алгоритм, який на кожному кроці знаходить найменший (або найбільший) елемент у невідсортованій частині масиву та міняє його місцями з першим елементом цієї частини. Цей процес повторюється доти, доки весь масив не буде відсортований.

Принцип роботи:

– пошук мінімального елемента – у невідсортованій частині масиву відшукується елемент з мінімальним значенням;

– обмін – знайдений мінімальний елемент міняється місцями з першим елементом поточної невідсортованої ділянки;

– перехід до наступної ділянки – перший елемент вважається відсортованим, і алгоритм переходить до наступної невідсортованої ділянки, яка тепер починається з другого елемента, і процес повторюється;

– алгоритм завершується, коли залишається лише один елемент, який не потребує сортування.

Приклад. Розглянемо масив `[1, 5, 3, 2, 4]`:

Крок 1. Найменший елемент `1`. Він вже на першій позиції, тому обміну не відбувається. Масив залишається `[1, 5, 3, 2, 4]`.

Крок 2. Розглядаємо підмасив `[5, 3, 2, 4]`. Найменший елемент `2`. Міняємо його місцями з 5. Масив стає `[1, 2, 3, 5, 4]`.

Крок 3. Розглядаємо підмасив `[3, 5, 4]`. Найменший елемент `3`. Він вже на своїй позиції, обміну не відбувається. Масив залишається `[1, 2, 3, 5, 4]`.

Крок 4. Розглядаємо підмасив `[5, 4]`. Найменший елемент `4`. Міняємо його місцями з 5. Масив стає `[1, 2, 3, 4, 5]`.

Крок 5. Залишився один елемент `5`, тому сортування завершено.

Сортування вибором має квадратичну часову складність $O(n^2)$, що робить його неефективним для великих масивів. Проте, він простий для розуміння та реалізації, що робить його гарним вибором для навчальних цілей або для сортування невеликих колекцій даних (лістинг 8.14).

Лістинг 8.14 – Сортування методом вибору (selection sort)

```
for (int i = 0; i < N - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < N; j++)
        if (arr[j] < arr[minIndex])
            minIndex = j;
    swap(arr[i], arr[minIndex]);
```

Кінець лістингу 8.14

8.3.3 Сортування вставками (insertion sort)

Сортування вставками (Insertion sort) – це простий алгоритм сортування, який послідовно будує відсортований масив шляхом вибору елементів з невідсортованої частини та вставки їх на правильне місце в уже відсортованій частині. Він працює, перебираючи масив і переміщуючи кожен елемент до його правильної позиції шляхом порівняння з попередніми елементами.

Принцип роботи:

– масив ділиться на дві частини: відсортовану (спочатку один елемент) і невідсортовану;

– на кожному кроці алгоритм бере перший елемент з невідсортованої частини;

– цей елемент порівнюється з елементами у відсортованій частині, рухаючись з кінця до початку;

– якщо елемент з відсортованої частини більший за поточний елемент, він зсувається на одну позицію вправо, щоб звільнити місце для вставки;

– коли знайдено елемент, менший за поточний, або досягнуто початку масиву, поточний елемент вставляється на звільнене місце;

– процес повторюється, доки всі елементи не будуть переміщені з невідсортованої частини до відсортованої.

Має квадратичну складність $O(n^2)$ в найгіршому випадку (коли масив відсортований у зворотному порядку), але в найкращому випадку (майже відсортований масив) його складність стає лінійною O(n).

Вимагає O(1) додаткової пам’яті для збереження поточного елемента.

Це «стійкий» алгоритм сортування, тобто однакові елементи не змінюють відносного порядку.

Ефективний для невеликих або майже відсортованих масивів (лістинг 8.15).

Лістинг 8.15 – Сортування вставками (insertion sort)

```
// Ідея: кожен елемент вставляється у відсортовану частину масиву.
for (int i = 1; i < N; i++) {
    int key = arr[i];
    int j = i - 1;
    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];
        j--;
    }
    arr[j + 1] = key;
}
```

Кінець лістингу 8.15

Перевірити чи масив впорядкований можна за допомогою лістингу 8.16.

Лістинг 8.16 – Перевірка, чи масив впорядкований

```
bool sorted = true;
```

```
for (int i = 0; i < N - 1; i++)
    if (arr[i] > arr[i + 1]) {
        sorted = false;
        break;
    }

if (sorted)
    cout << "Масив впорядкований за зростанням.";
else
    cout << "Масив не впорядкований.";
```

Кінець лістингу 8.16

Для сортування масивів у C++ можна використовувати функцію `std::sort` з бібліотеки `<algorithm>`, яка є найпростішим і найефективнішим способом (лістинг 8.17).

Лістинг 8.17 – Використання бібліотечної функції сортування

```
#include <iostream>
#include <algorithm>
int array[] = { 30, 50, 20, 10, 40 };
// Отримання розміру масиву
int length = sizeof(array) / sizeof(array[0]);
std::sort(array, array + length);
```

Кінець лістингу 8.17

8.4 Робота з двовимірними масивами

У пам’яті двовимірний масив розташовується *рядками підряд* (рядкова форма зберігання).

Наприклад:

```
A[2][3] = {
 {1, 2, 3},
 {4, 5, 6}
}
```

у пам’яті розташовується як послідовність: `1, 2, 3, 4, 5, 6`

Типові алгоритми опрацювання двовимірних масивів наведено в лістингах 8.18-8.29.

Лістинг 8.18 – Обчислення суми всіх елементів

```
int sum = 0;
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        sum += A[i][j];

cout << "Сума елементів = " << sum;
```

Кінець лістингу 8.18

Лістинг 8.19 – Пошук максимального і мінімального елементів

```
int max = A[0][0];
int min = A[0][0];

for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        if (A[i][j] > max) max = A[i][j];
        if (A[i][j] < min) min = A[i][j];
    }

cout << "Максимум = " << max << endl;
cout << "Мінімум = " << min << endl;
```

Кінець лістингу 8.19

Лістинг 8.20 – Обчислення суми елементів кожного рядка

```
for (int i = 0; i < M; i++) {
    int rowSum = 0;
    for (int j = 0; j < N; j++)
        rowSum += A[i][j];
    cout << "Сума рядка " << i << " = " << rowSum << endl;
}
```

Кінець лістингу 8.20

Лістинг 8.21 – Обчислення суми елементів кожного стовпця

```
for (int j = 0; j < N; j++) {
    int colSum = 0;
    for (int i = 0; i < M; i++)
```

```
        colSum += A[i][j];
    cout << "Сума стовпця " << j << " = " << colSum << endl;
}
```

Кінець лістингу 8.21

Лістинг 8.22 – Сума елементів головної діагоналі

```
int sum = 0;
for (int i = 0; i < M; i++)
    sum += A[i][i];

cout << "Сума головної діагоналі = " << sum;
```

Кінець лістингу 8.22

Лістинг 8.23 – Сума елементів бічної діагоналі

```
int sum = 0;
for (int i = 0; i < M; i++)
    sum += A[i][M - i - 1];

cout << "Сума бічної діагоналі = " << sum;
```

Кінець лістингу 8.23

Лістинг 8.24 – Транспонування матриці

```
//Транспонування – це операція обміну рядків і стовпців.
int B[N][M];
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        B[j][i] = A[i][j];
```

Кінець лістингу 8.24

Лістинг 8.25 – Підрахунок кількості від’ємних елементів

```
int count = 0;
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        if (A[i][j] < 0)
            count++;

cout << "Кількість від’ємних елементів: " << count;
```

Кінець лістингу 8.25

Лістинг 8.26 – Пошук заданого елемента

```
int x;
bool found = false;
cin >> x;

for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        if (A[i][j] == x) {
            cout << "Елемент знайдено на позиції [" << i << 
"][" << j << "]" << endl;
            found = true;
        }

if (!found)
    cout << "Елемент не знайдено.";
```

Кінець лістингу 8.26

Лістинг 8.27 – Обчислення середнього значення елементів

```
int sum = 0;
int count = M * N;

for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        sum += A[i][j];

cout << "Середнє значення = " << (double)sum / count;
```

Кінець лістингу 8.27

Лістинг 8.28 – Заміна елементів, що задовольняють умову

```
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
        if (A[i][j] < 0)
            A[i][j] = 0;
```

Кінець лістингу 8.28

Лістинг 8.29 – Сортування елементів у рядках або стовпцях

```
//Сортування кожного рядка за зростанням:
for (int i = 0; i < M; i++)
    for (int j = 0; j < N - 1; j++)
```

```
        for (int k = j + 1; k < N; k++)
            if (A[i][j] > A[i][k])
                swap(A[i][j], A[i][k]);
// Сортування кожного стовпця:
for (int j = 0; j < N; j++)
    for (int i = 0; i < M - 1; i++)
        for (int k = i + 1; k < M; k++)
            if (A[i][j] > A[k][j])
                swap(A[i][j], A[k][j]);
```

Кінець лістингу 8.29

Інформація про типові алгоритми опрацювання двовимірних масивів зведена в таблицю 8.3.

Таблиця 8.3 – Типові задачі з двовимірними масивами

№	Постановка задачі	Приклад алгоритму
1	Знайти суму всіх елементів	Подвійний цикл for
2	Знайти максимум (мінімум)	Порівняння всіх елементів
3	Знайти суму рядків/стовпців	Вкладений цикл
4	Знайти суму діагоналей	Індекси `A[i][i], A[i][N-i-1]`
5	Замінити від'ємні елементи	`if (A[i][j] < 0) A[i][j]=0`
6	Транспонування	`B[j][i] = A[i][j]`
7	Сортування рядків	Три вкладені цикли

Практичне застосування двовимірних масивів:

– табличні дані (таблиці студентів, результати вимірювань);

– матричні обчислення в інженерії;

– зображення в графічних програмах (матриця пікселів);

– ігрові поля (шахи, «хрестики-нулики»).

Контрольні питання

1. Як можна звернутися до елемента масиву за допомогою вказівника?
2. У чому полягає відмінність між доступом до елементів через індекси та через вказівники?
3. Як співвідносяться ім'я масиву та вказівник у мові C/C++?
4. Як можна організувати послідовний перегляд усіх елементів масиву за допомогою циклу?
5. Які типові алгоритми опрацювання одновимірних масивів ви знаєте?
6. Як реалізується алгоритм знаходження мінімального або максимального елемента в масиві?

7. Як реалізувати обчислення суми або середнього значення елементів масиву?

8. У чому полягає ідея алгоритму впорядкування елементів масиву за зростанням (сортування)?

9. Які існують основні алгоритми сортування?

10. Опишіть принцип роботи алгоритму сортування обміном (бульбашкове сортування).

11. Як організувати обробку елементів двовимірного масиву за допомогою вкладених циклів?

12. Як здійснюється звертання до елемента двовимірного масиву через індекси?

13. Як здійснюється звертання до елемента двовимірного масиву через вказівники?

14. Як можна передати двовимірний масив у функцію?

15. Наведіть приклади практичних задач, які доцільно розв’язувати з використанням двовимірних масивів.

ТЕМА 9. Опрацювання текстових даних

9.1 Рядок символів як масив елементів

У мові C++ рядок символів розглядається як масив символів типу `char`, елементи якого зберігаються послідовно в пам'яті. Кожен символ займає 1 байт, а в кінці рядка автоматично додається нульовий символ '`\0`', який позначає кінець рядка. Наприклад:

```
char str[] = "Hello";
```

У пам'яті це виглядає так:

H	e	l	l	o	\0

Тобто «`Hello`» – це фактично масив із 6 елементів (5 видимих + '\0'):

```
char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

На відміну від інших мов програмування у C++ не визначено спеціального типу для опрацювання рядків. Рядок символів розглядається як масив елементів типу `char`, який закінчується символом '`\0`' (нуль-символ) що є ознакою кінця рядка. Такі рядки називають ASCII-рядками.

Сталі типу рядок записують у лапках, наприклад, «`Луцький національний технічний університет`», «`студенти`», « » – рядок, що містить один символ-пропуск. У сталих рядках нуль-символ дописується автоматично.

Зауваження. Більшість компіляторів мови C++ автоматично додає нуль-символ у кінець рядка, тому зазначати його не обов'язково.

Рядок у стилі C – це масив символів типу `char`, який *закінчується* нульовим символом '`\0`'. Розмір масиву *на 1 більший*, ніж кількість видимих символів. Наприклад, «`Hi`» → фактично містить '`H`', '`i`', '`\0`' → 3 елементи.

9.2 Оголошення та ініціалізація символьних рядків

Масиви символів оголошують так:

```
char <назва рядка>[довжина рядка];
```

Під час оголошення символьного масиву необхідно до фактичної довжини рядка додати одиницю для нульового символу (але не у всіх компіляторах). Якщо масив символів оголошують й ініціалізують одночасно, то довжину можна не зазначати, компілятор визначить її. Оскільки рядки є масивами символів, то назва рядка є вказівником на його перший елемент (на перший символ).

Приклад. Розглянемо оголошення та ініціалізацію рядків

```
const char text1[] = "Ми вивчаємо програмування";
char slovo[] = "University";
char fraza1[11], fraza2[40];
```

Тут оголошено сталу `text1`, яка має значення "Ми вивчаємо програмування", символьні масиви: `slovo` (без зазначення розміру), `fraza1` (може містити до 10 символів) та `fraza2` (до 39 символів).

Символьний масив `slovo` ще можна оголосити так:

```
char slovo[11] = "University";
```

або так

```
char slovo[] = {‘U’, ‘n’, ‘I’, ‘v’, ‘e’, ‘r’, ‘s’,’I’,’t’, ‘y’, ‘\0’};
```

Тут потрібно вручну записати нуль-символ, інакше компілятор трактуватиме змінну `slovo` не як рядок, а як масив.

Отож, існує кілька способів оголошення та ініціалізації рядків (таблиця 9.1).

Таблиця 9.1 – Оголошення та ініціалізація рядків

Спосіб	Приклад	Особливість
Масив без ініціалізації	`char text[50];`	Виділяє пам’ять для 49 символів + ‘\0’
Масив із ініціалізацією	`char word[] = "Hello";`	Довжина визначається автоматично
Масив символів	`char word[6] = {‘H’,’e’,’l’,’l’,’o’,’\0’};`	Необхідно вказати ‘\0’ вручну
Сучасний рядковий тип	`string word = "Hello";`	Зручніший тип з бібліотеки <string>

9.3 Звертання до елементів символьних рядків

Оскільки рядок – це масив, доступ до окремого символа здійснюється *через індекс* (нумерація починається з 0):

```
char word[] = "World";
cout << word[0]; // W
cout << word[3]; // l
```

Можна змінювати символи:

```
word[0] = ‘M’;   // Тепер "Morld"
```

Не можна змінювати символи безпосередньо в константних літералах:

```
char *s= "Hello";
    s[0] = ‘M’;   // // помилка при зміні s[0]
```

Рядки можна опрацьовувати посимвольно за допомогою вказівників або назви масиву, наприклад, так:

```
for (int n = 0; n < 11; n++)
*(fraza1+n) = *(slovo+n);
cout<<fraza1;
```

Змінній `fraza1` надається значення "`University`" і ця фраза виводиться на екран. Інакше це можна зробити так:

```
for (int n = 0; n < 11; n++)
fraza1[n] = slovo[n];
cout<<fraza1;
```

9.4 Ведення/виведення символьних рядків

Увести весь масив символів можна ввести за допомогою команди `cin`:

```
cin >> <назва масиву>;
```

Якщо рядок даних містить символ пропуску, то команда `cin >>` зчитає дані лише до першого пропуску.

Вивести масив символів можна за допомогою `cout`:

```
cout<< <назва рядка>;
```

Приклад:

```
char name[20];
cout << "Введіть ім’я: ";
cin >> name;
cout << "Привіт, " << name << "!";
```

Нагадаємо, що `cin` читає до першого пробілу, тому фраза «`Світлана Лавренчук`» буде зчитана лише як «`Світлана`».

Щоб зчитати весь рядок до символу вводу, необхідно застосувати команду:

```
cin.get(<Назва рядка>, максимальна довжина рядка>);
```

Наприклад, `cin.get(fraza2, 40).`

Зчитати символ вводу можна так: `cin.get().`

Зчитати рядок разом із символом вводу можна одним із способів:

```
cin.get(fraza2,40);
cin.get();
cin.get(fraza2,40).get()
cin.getline(fraza2,40).
```

Функція `cin.getline()` зчитує рядок повністю, разом із пробілами:

```
char text[50];
cout << "Введіть текст: ";
cin.getline(text, 50);
cout << "Ви ввели: " << text;
```

Посимвольно вводити чи виводити елементи рядка можна за допомогою команд циклу `for` або `while`. Наприклад:

```
for (int n = 0; n < 11; n++)
cin >> *(fraza1 + n);
```

В кінці рядка необхідно поставити нуль-символ, тобто:

```
*(fraza1 +n + 1) = '\0';
```

У бібліотеці `conio.h` визначені стандартні функції введення-виведення рядків. Наприклад, `getc(), getchar()` зчитують по одному символу рядка, введеного з клавіатури, `putc()` та `putchar()` виводять окремі символи рядка тощо. У бібліотеці `stdio.h` описані функції для введення `gets()` та виведення `puts()` усього рядка. Детальніше по ці та інші функції написано у довідниках.

`string` – це клас із бібліотеки <`string`>, призначений для зручної роботи з текстовими рядками:

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

string s1 = "Hello";
string s2("World");
```

На відміну від масиву `char[], string` *автоматично керує пам'яттю*, дозволяє легко *додавати, порівнювати, копіювати* рядки.

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

int main() {
    string line;
    getline(cin, line);
    cout << "Рядок: " << line << endl;
}
```

`getline()` із типом `string` – найзручніший спосіб роботи з текстом у сучасному C++. Приклад:

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

int main() {
    string name;
    cout << "Введіть ім’я: ";
    getline(cin, name);
    cout << "Привіт, " << name << "!" << endl;
}
```

9.5 Бібліотечні функції для роботи зі символами та символьними рядками засобами мови C/C++

Функціонал для роботи зі символами та рядками в C/C++ реалізовано двома основними способами:

- через C-стиль рядків (символьні масиви з нульовим термінатором);
- через стандартний клас `std::string` в C++.

9.5.1 Робота з окремими символами

Функції, що приймають символ (`char` або `int`) і виконують перевірку його типу або зміну регістру (з бібліотеки <`cctype`> або <`ctype.h`>) наведено в таблиці 9.2.

Таблиця 9.2 – Функції для роботи з окремими символами

Функція	**Призначення**	**Приклад**
`isalpha(ch)`	Перевіряє, чи літера	`if (isalpha(ch))`
`isdigit(ch)`	Перевіряє, чи цифра	`if (isdigit(ch))`
`islower(ch)`	Чи малий регістр	`if (islower(ch))`
`isupper(ch)`	Чи великий регістр	`if (isupper(ch))`
`tolower(ch)`	Перетворює у нижній регістр	`ch = tolower(ch);`
`toupper(ch)`	Перетворює у верхній регістр	`ch = toupper(ch);`
`isspace(ch)`	Перевіряє, чи пробіл	`if (isspace(ch))`

9.5.2 Робота з рядками у стилі C

Функції, що працюють з рядками, представленими як масиви символів, що закінчуються нуль-символом (`\0`) (бібліотека <`cstring`> або <`string.h`>) наведено в таблиці 9.3.

Таблиця 9.3 – Функції для роботи з рядками у стилі C

Функція	**Призначення**	**Приклад**
`strlen(s)`	Довжина рядка (без ‘\0’)	`int n = strlen(str);`
`strcpy(dest, src)`	Копіює рядок	`strcpy(b, a);`
`strcat(dest, src)`	Додає один рядок до іншого	`strcat(a, b);`
`strcmp(a, b)`	Порівнює два рядки	`if (strcmp(a, b)==0)`
`strchr(s, ch)`	Пошук символа в рядку	`p = strchr(str, ‘a’);`
`strstr(s1, s2)`	Пошук підрядка	`p = strstr(text, "cat");`

9.5.3 Робота з рядками типу string

У C++ для роботи з рядками настійно рекомендується використовувати клас `std::string` (потрібно `#include <string>`). Він є безпечнішим, автоматично керує пам’яттю і має більш інтуїтивний об’єктно-орієнтований інтерфейс (таблиця 9.4).

Таблиця 9.4 – Функції для роботи з рядками у стилі C++

Операція	**Приклад**
Додавання	`s3 = s1 + s2;`
Порівняння	`if (s1 == s2)`
Витяг підрядка	`s.substr(pos, len)`
Пошук	`s.find("abc")`
Видалення	`s.erase(pos, len)`
Довжина	`s.length() або s.size()`
Доступ до символа	`s[i]`

Приклад програми (лістинг 9.1).

Лістинг 9.1 – Підрахунок кількості голосних у рядку

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

int main() {
    char text[100];
    cout << "Введіть рядок: ";
    cin.getline(text, 100);

    int count = 0;
    for (int i = 0; text[i] != ‘\0’; i++) {
        char ch = tolower(text[i]);
        if (ch==‘a’||ch==‘e’||ch==‘i’||ch==‘o’||ch==‘u’||
```

```
ch==‘a’||ch==‘e’||ch==‘и’||ch==‘i’||ch==‘o’||ch==‘y’)
            count++;
    }

    cout << "Кількість голосних: " << count << endl;
    return 0;
}
```

Кінець лістингу 9.1

Контрольні питання

1. Що таке символьний рядок у мові C++ і як він зберігається в пам’яті?
2. Яке призначення нульового символа ‘`\0`’ у символьних рядках?
3. Чим рядок у стилі C відрізняється від типу `string` у C++?
4. Як оголосити масив символів для зберігання тексту з 20 символів?
5. Як ініціалізувати символьний рядок під час оголошення?
6. Як звернутися до окремого символа в рядку? Наведіть приклад.
7. Яким чином можна змінити окремий символ у символьному рядку?
8. Як виконується введення рядка, що містить пробіли, у C++?
9. Яка різниця між операторами введення `cin >>` та функцією `cin.getline()`?
10. Як здійснюється виведення символьного рядка на екран?
11. Яке призначення бібліотеки <`cstring`> і які функції вона містить?
12. Як працюють функції `strlen()`, `strcpy()`, `strcat()` і `strcmp()`?
13. Які функції бібліотеки <`cctype`> використовуються для перевірки типів символів (літера, цифра тощо)?
14. Як виконати перетворення символу до верхнього або нижнього регістру?
15. Які переваги використання типу `string` у порівнянні з масивом символів типу `char[]`?

ТЕМА 10. Структури та об’єднання засобами мови С/С++

10.1 Поняття структури у мові С/С++. Оголошення та ініціалізація структур

Структура (`struct`*)* – це *користувацький тип даних*, який дозволяє об’єднати в одному об’єкті кілька змінних різних типів під одним ім’ям.

Структури використовують для опису *складних об’єктів* (наприклад, студент, товар, координата тощо).

Складові частини структури називають полями або елементами. Кожне поле структури має свій тип і своє ім’я.

Шаблон структури оголошується наступною синтаксичною конструкцією:

```
struct тег_структури {
тип_поля_1 ім’я_поля_1 ;
тип_поля_2 ім’я_поля_2;
тип_поля_k ім’я_поля_k;
};
```

тут `struct` – службове (ключове) слово, шо специфікує структуру;

тег_структури – ім’я, яким позначатимуть у програмі структури даної форми.

Список полів структури охоплюється фігурними дужками `{}`, після правої дужки `}` записується знак `;` (за умови, що після шаблону структури не оголошено структурних змінних).

Кожне поле структури описується як змінна – типом та іменем.

Тег (ярлик, ім’я) надається шаблону структури програмістом, щоб надалі в програмі можна було ідентифікувати структури вказаної форми і складу. Теги записуються як звичайні ідентифікатори мови С/С++.

Якщо в програмі використовується тільки одна структура, то її шаблон можна оголосити без тега:

```
struct { список полів структури };
```

Такий шаблон називають безіменним. Можна залишити безіменним *один* з усіх оголошених у програмі шаблонів структур.

Поля структури описуються послідовно. В оперативній пам'яті вони будуть записані саме в тому порядку, в якому були задані в шаблоні даної струкгури.

Тип поля може бути довільним простим або складеним типом. *Ім'я поля* є ідентифікатором змінної відповідного типу. В межах одного шаблону імена всіх полів повинні бути різними, але вони можуть збігатися з іменами полів інших структур чи з іменами змінних програми.

Для прикладу оголосимо шаблон структури, в яку можна буде записувати інформацію про студентів – учасників конкурсу:

```
struct student {
char name[40];
char grup[8];
long int number;
double rating; };
```

Тегом даної структури є ім'я `student`. Структура складається з чотирьох полів. Перші два поля `name` та `grup` є символьними рядками, в яких буде зберігатись інформація про прізвище та ім'я студента та його академгрупу. Поле `number` цілочислове, воно призначене для запису номера студентського квитка. Останнє поле `rating` має дійсний тип, у це поле буде заноситись середній бал успішності студента.

За своїм змістом шаблон структури – це опис нового типу, створеного програмістом. Для шаблону не виділяється місце в оперативній пам'яті, над ним не можна виконувати ніяких операцій. Шаблон слугує основою для оголошення змінних зі створеним структурним типом.

Застосовують дві форми *оголошення структурних змінних*:

– через посилання на попередньо оголошений шаблон;

– одночасно з оголошенням шаблону структури.

Перша форма передбачає, що шаблон структури вже оголошено. Тоді його тег спільно зі словом `struct` можна використовувати для створення змінних відповідного структурного типу.

Зокрема, використовуючи оголошений шаблон структури `student`, можна оголосити відповідні структурні змінні:

```
struct student studl, stud2;
struct student sarr[50];
```

У результаті даних оголошень в оперативній пам'яті буде виділено місце для двох змінних `studl` і `stud2` та для масиву `sarr`, що складається з 50-ти елементів.

Обидві змінні та всі елементи масиву `sarr` будуть мати однаковий розмір і склад полів, встановлений в оголошенні шаблону структури `student`.

Ще один приклад:

```
struct Point { int x; int y; };
Point p1;          // оголошення змінної типу Point
Point p2 = {3, 4}; // ініціалізація під час оголошення
```

Мова C/C++ дозволяє оголошувати структурні змінні одночасно з оголошенням шаблону структури. Наприклад, структурні змінні можна оголосити таким чином:

```
struct Point { int x; int y; } upleft, downright, polygon[8];
```

У наведеному прикладі, як і в попередньому, оголошено дві змінні та масив, але їх оголошення виконано разом із записом шаблону структури, призначеної для збереження координат точки площини. Кожна зі змінних `upleft` і `downright` (це можуть бути координати лівого верхнього і правого нижнього кутів прямокутника) буде складатись із двох цілочислових полів `x` та `y`. Масив `polygon` сформовано з восьми елементів, що мають тип **`struct`** `point` (елементи масиву можуть бути координатами вершин багатокутника).

Можна також робити оголошення з анонімною структурою:

```
struct {
```

```
    char name[20];
    int age;
} person1, person2;
```

В оголошеннях структурні змінні можна ініціалізувати, тобто присвоювати їх елементам початкові значення. Як і у випадку масиву, список значень, якими ініціалізується структура, вказується у фігурних дужках.

Приклад ініціалізації структурної змінної:

```
struct student {
char name[40];
char grup[8];
long int number;
double rating; };
struct student stud1 { "Іванов Андрій", "ІСТО-11",
1234567,95.23 );
```

Значення, якими ініціалізується структура, записують строго в тій самій послідовності, в якій вказані поля у шаблоні структури, а типи констант ініціалізаторів повинні бути сумісним з типами відповідних полів.

Можна виконувати часткову ініціалізацію – задавати значення лише декількох початкових елементів структури.

Якщо ініціалізується масив структур, то послідовно записуються значення, якими ініціалізується кожна зі структур, що входять до складу масиву, наприклад:

```
struct Point { int x; int y; };
struct Point outline[] = { {40, 45}, {86, 12}, {64, 23} };
```

Кількість елементів масиву `outline` буде визначатись кількістю ініціалізаторів-структур, для даного прикладу вона становитиме 3.

Якщо ініціалізуються всі поля структур, то внутрішні фігурні дужки можна опустити. Зокрема, ініціалізація масиву `outline` може бути такою:

```
struct Point outline[] = { 40, 45, 86, 72, 64, 23 };
```

Очевидно, що ця форма ініціалізації менш наочна, ніж перша, і це може стати причиною помилок у записі значень, якими ініціалізуються структури.

10.2 Розмір структури

Для кожної структурної змінної в оперативній пам’яті виділяється неперервна ділянка, обсяг якої дорівнює або більший за сумарний розмір усіх полів даної структури. Приміром, розмір змінної з типом **`struct`** `point` буде не меншим за 4 байти, а розмір змінної з типом **`struct`** `student` – не меншим за 60 байтів.

Реальний обсяг ділянки, яку займає в структурна змінна, залежить від прийнятого в даній системі способу вирівнювання даних. Залежмо від апаратно-програмних установок вирівнювання може виконуватись на межу слова, подвійного слова або не виконуватись взагалі.

У разі відсутності вирівнювання поля структури записуються підряд, інакше між полями можуть залишатись незайняті байти.

У програмі розмір структури треба визначати операцією `sizeof`. Нагадаємо, шо конструкція `sizeof(тип)` потребує дужок, а для виразу `sizeof змінна` у мові C/C++ дужки не обов’язкові. Наприклад, значенням виразу `sizeof (`**`struct`** `student)` буде розмір кожної змінної, що має тип **`struct`** `student`. Такий самий результат дадуть вирази: `sizeof best_stud, sizeof stucl`, а також `sizeof sarr[0]`.

Отож, розмір структури (`sizeof(`**`struct`**`)`) дорівнює *сумі розмірів її полів*, але з урахуванням *вирівнювання (alignment)* у пам’яті.

Приклад:

```
struct Example {
    char a;   // 1 байт
    int b;    // 4 байти (вирівнюється)
    double c; // 8 байт
};

cout << sizeof(Example); // може бути 16 байт
```

10.3 Операція присвоєння структур

Ще однією важливою властивістю структурних змінних є те, що вони розглядаються як звичайні змінні, а не як вказівники. Тому, якщо дві структурні змінні мають спільний шаблон, а отже займають однакові за обсягом ділянки в оперативній пам'яті, то можна присвоїти значення однієї структури іншій.

Реалізація присвоєння полягає в повному копіюванні ділянки структури, вказаної справа від знака присвоєння, в ділянку структурної змінної, вказаної зліва.

Оператор присвоєння `studl = best_stud;` перепише вміст усіх полів змінної `best_stud` у структуру `studl`, а оператор `polygon[5] = *outline;` скопіює в шостий елемеїгг масиву `polygon` вміст першої структури масиву `outline`. Між структурами одного типу можна використовувати операцію присвоєння:

```
Student s1 = {"Іван", 20, 89.5};
Student s2;
s2 = s1;  // копіюються всі поля
```

Операція *копіює* значення всіх полів *поелементно*.

Ще раз звернемо увагу на те, що в операції присвоєння обидві структури повинні мати однаковий тип, тобто спільний шаблон.

Інші операції (крім операції визначення адреси &, про яку мова йтиме далі) над цілими структурами не допускаються. Зокрема, не можна порівнювати структури. Всі дії щодо опрацювання структур виконуються над їх елементами.

10.4 Вкладені структури, масиви структур, вказівники на структури

10.4.1 Вкладення структур

Вже зазначалося, що поля структур можуть мати довільний тип, як простий, так і складений, зокрема поле може бути масивом, символьним рядком (як у структурі **`struct`** `student`) чи вкладеною структурою.

Для внутрішніх (вкладених) структур діють такі ж правила оголошення, як і для зовнішніх: можна окремо оголошувати шаблон структури або задавати його всередині зовнішньої структури разом з оголошенням відповідних змінних.

Наведемо приклад, що ілюструє вкладення структур. Нехай у програмі крім структури **struct** Point, опис якої було наведено раніше, оголошено також структуру зі шаблоном RGB, поля якої задають інтенсивність червоної, зеленої і синьої складових кольору зображення:

```
struct RGB {
int red, green, blue;
};
```

Ще один шаблон визначає структуру, яка описує коло:

```
struct circle {
struct point center;
unsigned int radius;
struct RGB color; };
```

Поля center і color цієї структури є вкладеними структурами, тобто їх розмір і склад задаються попередньо оголошеними шаблонами **struct** point і **struct** RGB.

Ініціалізацію відповідної структурної змінної можна виконати так:

```
struct circle bige = {{10,-40}, 180, {40, 18, 40}};
```

Першу пару значень, яка задає координати точки центра кола змінної bigc, та останню трійку, що задає інтенсивності RGB-складових кольору, записано в {}, аби підкреслити, що це вкладені структури, хоча в разі повної ініціалізації полів струкіурної змінної внутрішні дужки не є обов’язковими.

Ще один приклад вкладених структур:

```
struct Date {
    int day, month, year;
};

struct Student {
```

```
    string name;
    int age;
    Date birthday;  // вкладена структура
};

Student s = {"Анна", 19, {5, 10, 2005}};
cout << s.birthday.year; // 2005
```

10.4.2 Масиви структур

Якщо потрібно зберігати багато однотипних об’єктів – використовують масив структур.

```
struct Product {
    string name;
    double price;
};

Product shop[3] = {
    {"Хліб", 25.5},
    {"Молоко", 32.0},
    {"Цукор", 40.0}
};

cout << shop[1].name;  // Молоко
```

З наборів однотипних структур можна формувати масиви. Повторимо один із наведених раніше прикладів оголошення масиву структур:

```
struct Point outline[] = { {40, 45}, {86, 12}, {64, 23} };
```

Оскільки в оголошенні не вказано явно розмірність масиву `outline`, то вона встановлюється компілятором за кількістю проініціалізованих структур. Щоб отримати значення кількості елементів масиву програмно, треба скористатись виразом з операціями `sizeof`:

```
sizeof (outline) / sizeof (struct Point)
```

або

```
sizeof (outline) / sizeof (* outline)
```

Масиви структур повністю зберігають всі базові властивості масивів мови C/C++. До елементів такого масиву можна звертатись через індекси або через вказівники.

Зокрема, вираз `*outline` рівнозначний виразу `outline[0]` – його значенням є перша з усіх структур масиву `outline`. Відповідно вираз `outline[3]` або `*(outline+3)` визначає структуру, індекс якої в масиві дорівнює `3`.

10.4.3 Вказівники на структури

Оскільки структурні змінні не є вказівниками, а розглядаються як звичайні змінні програми, то до них можна застосовувати операцію визначення адреси `&`. Результатом операції буде адреса першого байта ділянки, яку займає в оперативній пам’яті структура-операнд.

Значення адреси структури можна присвоїти вказівнику, базовий тип якого збігається з типом структурної змінної. Якщо в програмі оголошено вказівник `struct Point *pstrp;` то йому можна присвоїти адресу кожної зі структурних змінних, що мають шаблон `struct Point`.

Наприклад, можна виконати таке присвоєння:

```
pstrp =&upleft;
```

У разі виконання наступного оператора:

```
pstrp = outline; /* або pstrp=&outline[0]; */
```

значення вказівника `pstrp` дорівнюватиме адресі першої структури з масиву `outline`. А в разі збільшення вказівника `pstrp++;` він пересунеться на наступний елемент масиву, тобто значенням `pstrp` стане адреса першого байта наступної структури з масиву `outline`.

Можна створювати *вказівники на структури* та звертатися до полів через оператор `->`:

```
Student s = {"Олег", 21, 90};
Student *p = &s;

cout << p->name;  // те саме, що s.name
```

```
p->grade = 95;    // зміна поля через вказівник
```

10.5 Звертання до елементів структур

Мова C/C++ має дві операції, призначені для звертання до елементів (полів) структури:

– операцію «крапка», яку позначають знаком `.` і застосовують у звертаннях до поля структури, заданої через змінну;

– операцію «стрілка», яку позначають парою знаків `->` (мінус і більше) та використовують для звертання до поля структури через вказівник на цю структуру.

Розглянемо детальніше обидві форми звертання.

10.5.1 Операція «крапка»

Виділення поля структури, заданої через змінну, виконує така синтаксична конструкція:

```
ім’я_структурної_змінної.ім’я_поля
```

Цей вираз називають уточненим іменем елемента структури. Його значенням є значення змінної, яка формує відповідне поле структури. Наприклад, звертання до полів структури `best_stud`, оголошення та ініціалізацію якої ми наводили раніше, дадуть такі результати:

– `best_stud.name` – рядок із 40-а символів, у який занесено стрічку «Іванов Андрій»;

– `best_stud.number` – число з типом `long int`, що має значення 1234567;

– `best_stud.rating` – дійсне число з типом `double`, що дорівнює 95.23.

Поля структури повністю зберігають властивості даних свого типу, над ними можна виконувати всі операції, що допускаються для цього типу. Зокрема, щоб змінити значення елементів структури `best_stud`, треба враховувати особливості типу кожного з її полів.

Наприклад, тут використовуємо звичайне присвоєння, оскільки поле `rating` є числовим:

```
best_stud.rating = newrating; /* зміна поля rating */
```

Тут застосовуємо функцію копіювання стрічок, оскільки змінюємо значення цілого символьного рядка:

```
strcpy(best_stud.grup, newgrupname); /* зміна поля grup */
```

Хоча за допомогою операції присвоєння можна повністю переписати вміст однієї структури в іншу (включаючи всі поля-масиви і поля-рядки), окремо до полів-масивів і полів-рядків застосовувати присвоєння не можна, бо їх уточнені імена залишаються константними вказівниками на перші елементи цих масивів.

Обидві операції виділення елементів структури: `.` та `->` мають однаковий найвищий рівень старшинства та асоціативність зліва направо, що є визначальним для формування виразів, у які входять елементи структур.

Значенням виразу

```
*best_stud.name /* перша літера name */
```

є перша літера рядка, в який записано прізвище студента.

Операція виділення поля `name`, позначена крапкою, має вище старшинство і виконується першою, а операція розадресації `*` застосовується до уточненого імені рядка і повертає його перший символ (літеру ‘`I`’).

Відповідно значенням виразу

```
best_stud.name[11]) /* дванадцята літера name */
```

є літера ‘A’ – символ з індексом 11 із рядка `name`. В даному випадку операції виділення поля структури `.` та виділення елемента масиву `[ ]` мають однакове старшинство, тому виконуються згідно з правилами асоціативності зліва направо, тобто спочатку виділяється рядок, а потім відбувається звертання до символа, що має індекс 11.

Таке саме значення матиме вираз

```
*(best_stud.name+ll) /* дванадцята літера name */
```

який є адресною формою звертання до елемента масиву.

Відзначимо принципову відмінність виразів `studl.grup[3]` та `sarr[3].grup` Обидва вирази звертаються до поля `grup` структур, що мають шаблон `student`. Але значенням першого виразу є четверта за порядком літера найменування групи з даних структури `stud1`, а значенням другого виразу є весь символьний рядок `grup` (точніше, вказівник на його перший символ), тобто другий вираз повертає найменування групи четвертого студента з масиву `sarr`.

Якщо поле структури є вкладеною структурою, то для звертання до полів внутрішньої структури операцію «крапка» застосовують кількаразово, відповідно до глибини вкладення структур.

Наприклад, щоб виділити координати центра кола, параметри якого задаються структурою `bigc`, описаною в попередньому праграфі, треба використати конструкції:

`bigc.center.x` – горизонтальна координата центра кола, її значення після ініціалізації `big` дорівнює `10`;

`bigc.center.y` – вертикальна координата центра кола, шо має значення 40.

Оператор `bigc.color.green *= 2;` який звертається до поля `green` внутрішньої структури `color`, збільшить удвічі інтенсивність зеленої складової кольору кола.

Коли звертаються до полів структури, яка є елементом масиву структур (таким, наприклад, як описані раніше `outline` чи `sarr`), то для доступу до елемента масиву застосовують як індексну, так і вказівникову форму звертання.

Обидва вирази: `sarr[k].number` та `(*(sarr+k)).number` рівнозначні, хоча очевидно, що другий вираз малонаочний і важчий для сприйняття. Зовнішні дужки в записі `(*(sarr+k))` необхідні з огляду на старшинство операції «крапка» щодо операції розадресації *.

10.5.2 Операція «стрілка»

Операція `->` спрощує звертання до полів структур, які виконуються через адреси цих структур або вказівники на структури.

Синтаксис її такий:

```
вказівник_на_структуру->ім’я_поля
```

Вираз sarr+k з попереднього прикладу є адресою k-ї структури з масиву sarr, тому для звертання до поля number даної структури можна використати операцію «стрілка»:

```
(sarr+k)->number
```

Усі записані далі вирази матимуть однакове значення – першу літеру прізвища першого студента з масиву sarr:

```
sarr[0].name[0]
*sarr[0].name
*(*sarr).name
sarr-> name[0]
* (&sarr[0])->name
*sarr->name
```

Вказівникова форма звертання особливо ефективна, коли для роботи з масивом структур використовують зовнішні вказівники, базовий тип яких збігається з типом структур – елементів масиву.

Отже, для доступу до полів структури використовується оператор крапки (.) або оператор стрілки (->) для вказівників (таблиця 10.1).

Таблиця 10.1 – Способи доступу до полів структури

Синтаксис	**Приклад**	**Пояснення**
об’єкт.поле	s1.age	доступ до поля без вказівника
вказівник->поле	p->age	доступ через вказівник

Розглянемо програмку, що демонструє приклад роботи зі структурами (лістинг 10.1).

Лістинг 10.1 – Приклад роботи зі структурами

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

struct Student {
```

```
    string name;
    int age;
    double grade;
};

int main() {
    Student group[3] = {
        {"Іван", 19, 85.5},
        {"Марія", 20, 90.0},
        {"Олег", 18, 78.0}
    };

    for (int i = 0; i < 3; i++) {
        cout << group[i].name << " – " << group[i].grade << endl;
    }
}
```

Кінець лістингу 10.1

10.6 Перейменування типів

До складу конструктивних елементів мови C/C++ входить спеціальна декларація `typedef`, яка дає програмістові змогу надавати власні імена типам даних програми.

Наприклад, за допомогою **`typedef`** можна перейменувати тип `unsigned int` на коротшу форму запису – `UINT`:

```
typedef unsigned int UINT;
```

Надалі найменування типу UINT можна використовувати в усій програмі там, де треба оголошувати змінні з типом **`unsigned int`**:

```
UINT max, min, numb, k;
```

Загальна конструкція декларації **`typedef`** така:

```
typedef тип користувацьке_ім’я_типу;
```

Всередині **typedef** користувацьке_ім’я_типу записується в тому місці, де для звичайного оголошення вказується ім’я змінної. Якщо в програмі задекларовано:

```
typedef char STRING(300];
```

то тип STRING буде масивом, що складається із 300 елементів-символів.

Цей тип можна використовувати для оголошення відповідних символьних рядків, а також типів, похідних від них.

Наприклад:

```
STRING stl, st2, stnew[25];
```

За цим оголошенням `stl` і `st2` будуть символьними рядками, кожен з яких може містити до 300 символів, а `stnew` – двовимірним масивом, що складається із 25-ти символьних рядків з типом STRING.

Декларація **typedef** *не ств*орює нового типу, а *тільки змінює ім’я існуючого* стандартного чи користувацького типу або іменує оголошений у декларації тип.

Використання декларації **typedef** сприяє підвищенню наочності імен типів даних та скорочує їх запис. Це передусім стосується таких типів як структури, об’єднання і переліки, для яких **typedef** можна використовувати одночасно з оголошенням шаблону.

Наприклад, перейменувавши шаблон структури, що описує хімічний елемент:

```
typedef struct chemical_element {
char name[16];
int number;
char abravir[4];
double mass; } ChemEl;
```

надалі зможемо використовувати ім’я структурного типу `ChemEl` у всіх наступних оголошеннях:

```
ChemEl elemset[100], *pelem;
```

Крім наочності та простоти іменування типів, **typedef** надає програмістові змогу створювати машинонезалежні мобільні типи даних. Насамперед це стосується тих типів, параметри яких залежать від апаратних особливостей комп'ютера. Якщо такому типові надати певне typedef-ім'я (прикладом може слугувати декларація типу UINT) і використати це ім'я всюди в тексті програми, то в новому середовищі достатньо лише внести зміни в рядок декларації typedef, щоб встановити потрібні параметри для перейменованого типу.

Наприклад, щоб тип UINT був двобайтовим цілим типом на комп'ютерах, для яких int є чотирибайтовим, треба в його декларацію додати слово short:

```
typedef unsigned short int UINT;
```

Щоб зручно використовувати структуру як окремий тип, застосовують typedef або using:

```
typedef struct {
    int day, month, year;
} Date;
```

або у стилі C++:

```
using Date = struct { int day, month, year; };
```

Тоді можна писати просто:

```
Date today = {13, 10, 2025};
```

10.7 Об'єднання (union)

Об'єднання – це особливий тип даних, який дає змогу записувати в одну і ту ж встановлену ділянку оперативної пам'яті дані різних типів і розмірів. Таким чином створюється ділянка пам'яті спільного користування, до якої можна звертатись різними способами через єдину змінну, що має тип об'єднання.

Оголошення об'єднання подібне до оголошення структури: задається шаблон об'єднання та перелічуються відповідні змінні. Так само, як і для

структур, шаблон об’єднання можна оголошувати окремо або одночасно з оголошенням змінних.

У разі автономного оголошення шаблону об’єднання застосовують синтаксичну конструкцію, аналогічну до шаблону структури, але починають її ключовим словом **union**:

```
union тег_об’єднання {
тип_поля_1 ім’я_поля_1;
тип_поля_2 ім’я_поля_2;
тип_поля_k ім’я_поля_k; };
```

Тег_об’єднання ідентифікує дане об’єднання, а список полів задає перелік даних, які можна заносити в це об’єднання.

Як і у випадку структур, змінні з типом об’єднання можна оголошувати спільно з оголошенням шаблону або пізніше, використовуючи тип **union** тег_об’єднання.

Обсяг ділянки пам’яті, яка виділяється для кожної змінної, що має тип об’єднання, визначається розміром найдовшого поля даного об’єднання.

У декларації **typedef** можна сумістити оголошення шаблону об’єднання та його найменування. Наведемо приклад оголошення іменованого об’єднання:

```
typedef union demo_union {
double x;
char ch;
int a; } UNDEM;
```

Якщо тепер оголосити змінну z, використовуючи задекларований тип UNDEM:

```
UNDEM z;
```

то така змінна буде об’єднанням із шаблоном demo_union, тобто всі три її поля: x, ch та a будуть користуватись спільною ділянкою оперативної пам’яті.

Обсяг ділянки, яку займає змінна z, дорівнює розміру її найдовшого поля. У нашому випадку найдовше поле x, воно має тип double і займає 8 байтів.

Відзначимо, що всі три поля змінної z мають однакове нульове зміщення відносно адреси початку ділянки, яку займає z.

Якщо молодший байт даних має адресу, меншу, ніж старший байт, то поля об'єднання накладаються одне на інше в області молодших адрес. Байт, який займає поле ch, збігається з молодшим байтом поля a та з наймолодшим байтом поля змінної x.

Оголошуючи змінну з типом об'єднання, можна відразу проініціалізуватн цю змінну, тобто записати у її ділянку певне початкове значення. Звернемо увагу, шо ініціалізувати об'єднання можна тільки значенням, сумісним з типом першого поля цього об'єднання (для змінної z це тип double).

Як і у випадку масивів та структур, список значень, якими ініціалізується об'єднання (навіть якщо це буде тільки одне значення), записується у фігурних дужках. До змінних з типом об'єднання можна застосовувати всі ті ж самі операції, що й до структурних змінних: присвоєння (тобто копіювання цілого об'єднання), визначення адреси змінної, виділення окремих елементів (полів) об'єднання.

Для звертання до елементів об'єднання використовують такі ж синтаксичні конструкції на основі операцій «крапка» . та «стрілка» ->, як і в разі звертання до елементів структур:

```
ім'я_змінної_об'єднання.ім'я_поля
вказівник_на_об'єднання->ім'я_поля
(*вказівник_на_об'єднання).ім'я_поля
```

Наприклад, якшо виконати присвоєння

```
z.ch='A';
```

то у поле ch змінної z оголошеної вище, буде занесено код символа 'A' (65). Тим самим наймолодші байти полів a та x набудуть значення 65.

У разі наступного присвоєння:

```
(&z)->a=0x3c32; /* те ж саме, що й z.a=>0x3c32 */
```

зміняться два молодші байти об'єднання z. Тепер значенням виразу z.ch буде символ з шістнадцягковим кодом 0x32, тобто символ '2'.

Отже, об’єднання (union) – це тип, у якому всі поля розміщуються в одній і тій самій ділянці пам’яті. Тобто в один момент часу можна використовувати *тільки одне поле.*

Приклад:

```
union Data {
    int i;
    float f;
    char str[10];
};

Data d;
d.i = 5;
cout << d.i;  // 5

d.f = 3.14;   // тепер змінено те саме місце в пам’яті
```

Розмір union дорівнює розміру найбільшого поля.

10.8 Поля бітів (bit fields)

Поля бітів (їх називають також бітові поля) – це особливий вид елементів структур та об’єднань. Такі поля складаються з послідовних бітів, записаних у дане цілого типу.

Поля бітів застосовують, коли необхідно мати доступ до окремих бітів всередині байта або слова даних.

Найчастіше така потреба виникає у процесі керування апаратними засобами комп’ютера, наприклад, у разі звертання до регістрів відеоадаптера.

Іншим поширеним застосуванням бітових полів є стиснення даних для компактного збереження інформації, насамперед у випадках, коли дані мають булів тип («так» або «ні»), тому для їх збереження достатньо одного біта.

Оголошення поля бітів виконується через синтаксичну конструкцію:

```
тип_поля ім’я_бітового_поля : розмір_ поля;
```

де `тип_поля` може бути тільки типом `int` з допоміжними специфікаторами `unsigned` або `signed`;

`ім’я_бітового_поля` – звичайний ідентифікатор, а `розмір_поля` – ціле число, яке задає ширину поля в бітах.

Поля бітів використовуються для *економії пам’яті*, коли потрібно зберігати дані, що займають кілька бітів (наприклад, прапорці або коди станів).

Приклад:

```
struct Flags {
    unsigned int isVisible : 1;
    unsigned int isActive  : 1;
    unsigned int isError   : 1;
};
```

Тут кожне поле займає лише *1 біт*, тому вся структура може міститися в одному байті.

Контрольні питання

1. Що таке структура у мові C/C++ і для чого вона використовується?
2. Як оголосити структуру та які елементи вона може містити?
3. Як здійснюється ініціалізація структур під час оголошення?
4. Як звертатися до елементів структури у програмі?
5. Чим відрізняється звертання до елементів структури через змінну та через вказівник?
6. Як оголосити масив структур і звертатися до його елементів?
7. Що означає вкладена структура і як отримати доступ до її полів?
8. Як працює операція присвоєння між структурами одного типу?
9. Як визначити розмір структури та чому він може бути більшим за суму розмірів її полів?
10. Як створити вказівник на структуру та використовувати оператор ->?
11. Для чого використовується оператор `typedef` або ключове слово `using` у контексті структур?
12. Що таке об’єднання (`union`) і чим воно відрізняється від структури?
13. Як визначається розмір об’єднання?
14. Що таке поля бітів (`bit fields`) і де вони застосовуються?
15. Наведіть приклад практичного використання структур або об’єднань у програмі.

ТЕМА 11. Робота з файлами

11.1 Поняття файла та файлового потоку

Файл (File) у контексті програмування – це іменована область на зовнішньому носії інформації (наприклад, жорсткому диску, SSD), яка містить послідовність байтів. Файл використовується для постійного (довготривалого) зберігання даних.

Для обміну інформацією між програмою та файлом використовується файловий потік (`file stream`).

Файловий потік – це абстракція, яка використовується для здійснення операцій введення/виведення (`I/O`) між програмою та файлом. Потік являє собою послідовність даних, що тече від джерела (файл, клавіатура) до приймача (файл, екран).

У C++ потоки реалізовані класами з бібліотеки `<fstream>` (`std::ifstream` для вхідного, `std::ofstream` для вихідного, `std::fstream` для обох).

У C потоки представлені структурою `FILE*` (потрібно `<stdio.h>`).

У мові C++ для роботи з файлами використовуються класи бібліотеки `<fstream>`:

– `ifstream` – для *читання* з файлу (input file stream);

– `ofstream` – для *запису* у файл (output file stream);

– `fstream` – для *читання та запису* (file stream).

Підключення бібліотеки:

```
#include <fstream>
```

Приклад наведено в лістингу 11.1.

Лістинг 11.1 – Приклад роботи з файлами засобами мови C++

```
#include <fstream>
#include <iostream>
```

```
using namespace std;

int main() {
    ofstream fout("data.txt");  // відкриття файлу для запису
    fout << "Hello, file!";
    fout.close();              // закриття файлу

    ifstream fin("data.txt");   // відкриття файлу для читання
    string text;
    fin >> text;
    cout << text;
    fin.close();
}
```

Кінець лістингу 11.1

11.2 Типи файлів

Файли поділяють на:

1) *Текстові файли* зберігають дані у вигляді *символів* (ASCII або Unicode). Їх властивості:

– кожен елемент – літера, цифра, пробіл або символ нового рядка;

– дані (навіть числа) зберігаються у вигляді їх *символьних* представлень;

– можна переглянути у текстовому редакторі;

– приклади: `.txt, .csv, .cpp.`

```
ofstream fout("info.txt");
fout << "Name: Ivan\nAge: 20";
fout.close();
```

Перевага: легко читати людиною та переносити між різними операційними системами.

2) *Бінарні файли* зберігають інформацію у внутрішньому (двійковому) форматі. Їх властивості:

– дані записуються без перетворення у символи;

– менший розмір, швидше зчитування;

– використовується для збереження структур, об’єктів, масивів.

```
ofstream fout("nums.bin", ios::binary);
int x = 100;
fout.write((char*)&x, sizeof(x));
fout.close();
```

Перевага: ефективніше використання дискового простору та швидше читання/запис, оскільки не потрібна конвертація (форматування) даних.

Недолік: складніше переносити між різними архітектурами (через різний порядок байтів).

11.3 Режими роботи з файлами

Режим роботи (таблиця 11.1) визначає, що програма може робити з файлом (читати, писати) і як слід поводитися, якщо файл не існує або вже містить дані.

Таблиця 11.1 – Режими роботи з файлами

Режим		**Опис**	**Дія, якщо файл не існує**
C	**C++**		
`"r"`	`std::ios::in`	*Читання* (Read). Потік відкривається для читання даних з початку файлу.	*Помилка* (файл повинен існувати).
`"w"`	`std::ios::out`	*Запис* (Write). Потік відкривається для запису.	*Створюється*. Якщо існував, його *вміст знищується* (перезапис).
`"a"`	`std::ios::app`	*Додавання* (Append). Потік відкривається для запису. Всі нові дані додаються в **кінець** файлу.	*Створюється*.
`"r+"`	`std::ios::in | std::ios::out`	*Читання та запис*. Файл повинен існувати.	Помилка.
`"w+"`	`std::ios::out | std::ios::in`	*Читання та запис*. Вміст файлу знищується.	*Створюється*.

Приклад відкриття файлу з кількома режимами:

```
ofstream fout("log.txt", ios::out | ios::app);
```

Для бінарних файлів: до текстових режимів (наприклад, `"r"`, `"w"`) у C додається літера `b` (наприклад, `"rb"`, `"wb"`). У C++ використовується додатковий прапор `std::ios::binary`.

11.4 Функцій для роботи з файлами

11.4.1 Функції мови С (бібліотека <stdio.h>)

Функції мови С для роботи з файлами наведено в таблиці 11.2.

Таблиця 11.2 – Функції мови С для роботи з файлами (бібліотека <stdio.h>)

Функція	**Призначення**
`FILE *fopen(const char *filename, const char *mode)`	*Відкриває* файл у заданому режимі. Повертає вказівник на структуру `FILE`, або `NULL` у разі помилки.
`int fclose(FILE *stream)`	*Закриває* потік, записує буферизовані дані на диск і звільняє ресурси.
`int fprintf(FILE *stream, const char *format, ...)`	*Запис* форматованих даних у текстовий файл.
`int fscanf(FILE *stream, const char *format, ...)`	*Читання* форматованих даних з текстового файлу.
`size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream)`	*Запис* блоку даних (бінарний запис).
`size_t fread(void *ptr, size_t size, size_t count, FILE *stream)`	*Читання* блоку даних (бінарне читання).

Приклад роботи з текстовим файлом засобами мови С наведено в лістингу 11.2, приклад роботи з бінарним байлом – в лістингу 11.3.

Лістинг 11.2 – Приклад роботи з текстовим файлом засобами мови С

```c
#include <stdio.h>

int main() {
    FILE *f;

    // Запис у файл
    f = fopen("names.txt", "w");
    fprintf(f, "Anna\nOleh\nMarta\n");
    fclose(f);

    // Читання з файлу
    char name[50];
    f = fopen("names.txt", "r");
    while (fgets(name, 50, f) != NULL) {
        printf("%s", name);
    }
    fclose(f);

    return 0;
```

```
}
```

Кінець лістингу 11.2

Лістинг 11.3 – Приклад роботи з бінарним файлом засобами мови C

```c
#include <stdio.h>

int main() {
    int arr[3] = {10, 20, 30};
    int readArr[3];

    // Запис у бінарний файл
    FILE *f = fopen("data.bin", "wb");
    fwrite(arr, sizeof(int), 3, f);
    fclose(f);

    // Зчитування з бінарного файлу
    f = fopen("data.bin", "rb");
    fread(readArr, sizeof(int), 3, f);
    fclose(f);

    for (int i = 0; i < 3; i++) {
        printf("%d ", readArr[i]);
    }

    return 0;
}
```

Кінець лістингу 11.3

11.4.2 Класи мови C++ (бібліотека <fstream>)

C++ використовує об’єктно-орієнтований підхід через спеціалізовані класи потоків (таблиця 11.3).

Таблиця 11.3 – Функції (методи) мови C++ для роботи з файлами

Клас / Метод	**Призначення**
`std::ofstream`	Потік для *запису* у файл (Output File Stream).
`std::ifstream`	Потік для *читання* з файлу (Input File Stream).
`std::fstream`	Потік для *читання та запису* (File Stream).
`stream.open(filename, mode)`	*Відкриває* файл. Режим задається прапорами `std::ios::....`
`stream.close()`	*Закриває* потік.
`stream << data`	Оператор *запису* (використовується для текстового запису).

`stream >> data`	Оператор *читання* (використовується для текстового читання).
`stream.write(ptr, size)`	*Запис* блоку даних (бінарний).
`stream.read(ptr, size)`	*Читання* блоку даних (бінарне).
`stream.eof()`	Перевіряє, чи досягнуто *кінець файлу* (End Of File).

Приклад роботи з текстовим файлом засобами мови C++ наведено в лістингу 11.4, приклад роботи з бінарним байлом – в лістингу 11.5.

Лістинг 11.4 – Приклад роботи з текстовим файлом засобами мови C++

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

int main() {
    // Запис
    ofstream out("names.txt");
    out << "Anna\nOleh\nMarta\n";
    out.close();

    // Читання
    ifstream in("names.txt");
    string name;
    while (getline(in, name)) {
        cout << name << endl;
    }
    in.close();
}
```

Кінець лістингу 11.4

Лістинг 11.5 – Приклад роботи з бінарним файлом засобами мови C++

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

int main() {
    int arr[3] = {10, 20, 30};

    // запис у двійковий файл
    ofstream out("data.bin", ios::binary);
    out.write((char*)arr, sizeof(arr));
    out.close();
```

```
    // зчитування з файлу
    int readArr[3];
    ifstream in("data.bin", ios::binary);
    in.read((char*)readArr, sizeof(readArr));
    in.close();

    for (int x : readArr) cout << x << " ";
}
```

Кінець лістингу 11.5

Робота з файлами потребує *контролю відкриття та закриття* файлів для уникнення втрати даних.

Контрольні питання

1. Що таке файл у контексті програмування мовою C/C++?
2. Яке призначення файлового потоку у мові C++?
3. Як у мові C відкривається файл для читання та запису?
4. Які типи файлів підтримуються в C/C++? Наведіть приклади.
5. Чим відрізняються текстові та бінарні файли?
6. Які режими відкриття файлів існують у мові C?
7. Які режими відкриття файлів використовуються у мові C++?
8. Як відбувається процес відкриття та закриття файлу в обох мовах?
9. Які функції використовуються для запису даних у файл у мові C?
10. Як здійснюється читання даних із файлу в мові C?
11. Які методи або функції використовуються для запису та зчитування файлів у мові C++?
12. Як перевірити, чи файл успішно відкрився, у мовах C і C++?
13. Як виконати запис і зчитування структур або масивів у бінарному режимі?
14. Для чого використовуються функції `fseek()` і `ftell()` у мові C?
15. Які типові помилки виникають при роботі з файлами і як їх уникнути?

СПИСОК РЕКОМЕНДОВАНИХ ДЖЕРЕЛ

1. Лабораторний практикум з програмування мовою C/C++ : навч. посіб. для студ. тех. спец. закл. вищ. освіти I–IV рівн. акредит./ П. А. Пех, С. В. Лавренчук, М. В. Делявський, С. В. Гринюк. Луцьк, 2020. 228 с.

2. Рудий Т. В., Паранчук Я. С., Сеник В. В. Алгоритмізація та програмування. Частина 1. Структурне програмування: навчальний посібник. Львів: Львівський державний університет внутрішніх справ, 2023. 240 с.

3. Рудий Т. Алгоритмізація та програмування. Частина 2. Модульне програмування: навчальний посібник / Т. Рудий, Я. Паранчук, В. Сеник. Львів: Львівський державний університет внутрішніх справ, 2024. 176 с.

4. Ковалюк Т. В. Алгоритмізація та програмування: Підручник. Львів: «Магнолія 2006», 2024. 400 с.

5. Фратавчан В. Г., Фратавчан Т. М., Лазорик В. В. Алгоритмізація та програмування, навчальний посібник для закладів вищої освіти. ЧНУ, 2022. 286 с.

6. Васильєв О. М. Програмування в Python. Теорія і практика : навч. посіб. Київ : Видавництво Ліра-К, 2023. 462 с.

7. Бхаргава А. Грокаємо алгоритми. Ілюстрований посібник для програмістів і допитливих. ArtHuss, 2024. 256 с.

ІНФОРМАЦІЙНІ РЕСУРСИ

1. C++ Tutorials. URL: http://www.cplusplus.com/doc/tutorial/ (дата звернення: 25.05.2025).

2. Visualizing Algorithms. URL: https://bost.ocks.org/mike/algorithms/ (дата звернення: 29.05.2025).

3. Інструмент для створення блок-схем URL: https://app.diagrams.net/ (дата звернення: 25.05.2025).

4. Он-лайн компілятор URL: https://www.onlinegdb.com/ (дата звернення: 25.05.2025).

А45 **Алгоритмізація та програмування**: конспект лекцій (частина перша) для здобувачів першого (бакалаврського) рівня вищої освіти освітньої програми «Інформаційні системи та технології охорони і безпеки» галузь знань F Інформаційні технології спеціальності F6 Інформаційні системи та технології денної та заочної форм навчання / уклад. С .В. Лавренчук. Луцьк: ЛНТУ, 2025. 192 с.

Конспект лекцій (частина перша) з дисципліни «Алгоритмізація та програмування» складений відповідно до діючої програми курсу.

Призначений для здобувачів вищої освіти спеціальності Інформаційні системи та технології освітньої програми «Інформаційні системи та технології охорони і безпеки».

Комп’ютерний набір С.В. Лавренчук

Редактор С.В. Лавренчук

Підп. до друку «___» _____ 2025р.
Формат 60х84/16. Папір офс. Гарнітура Таймс.
Ум. друк. арк. ____. Тираж 10 прим. Зам. ____

Луцький національний технічний університет
43018, м. Луцьк, вул. Львівська, 75