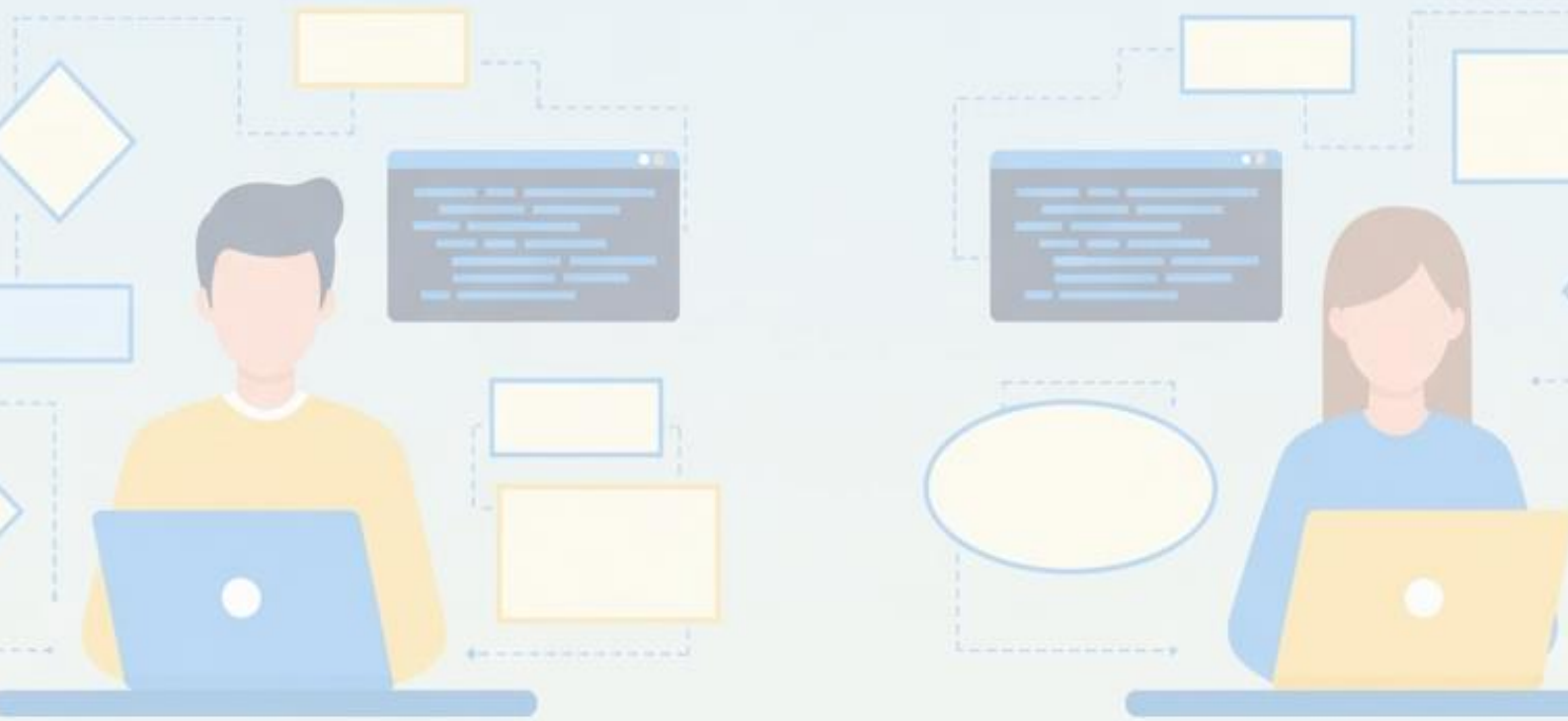
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Золотухіна О. А.



Програмування C++

Навчальний посібник

Київ – 2024

УДК 004.42:004.41

Рецензенти:

Жебка В.В., д.т.н., професор, завідувач кафедри технологій цифрового розвитку Державного університету інформаційно-комунікаційних технологій.

Шушура О.М., доктор технічних наук, доцент, професор кафедри цифрових технологій в енергетиці Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського».

Затверджено Вченою радою Навчально-наукового інституту інформаційних технологій Державного університету телекомунікацій
(протокол №10 від 13.05.2024 р.)

Автор: Золотухіна О.А.

Програмування С++. Навчальний посібник. Київ: ННІІТ ДУІКТ, 2024. –107 с.

Навчальний посібник призначено для студентів першого (бакалаврського) рівня вищої освіти спеціальності 121-«Інженерія програмного забезпечення», що вивчають дисципліну «Програмування С++». До посібника включено теоретичний матеріал для самостійного опанування основ створення програм мовою С++, приклади програмного коду для розв’язання типових елементарних практичних задач, завдання для самостійного опрацювання та контрольні питання. Тематика розділів включає алгоритмічні основи, базові компоненти мови С++, роботу з різними типами та структурами даних та структурування програм.

Посібник може бути використаний студентами спеціальності 121-«Інженерія програмного забезпечення» та інших спеціальностей галузі 12-«Інформаційні технології» для підготовки до практичних та лабораторних робіт, поточного та підсумкового оцінювання, організації самостійної роботи при вивченні дисциплін «Програмування С++», «Основи програмування» та інших дисциплін, в яких використовується мова С++, а також всіма, хто самостійно вивчає програмування.

ЗМІСТ

ВСТУП

Навчальна дисципліна «Програмування С++» вивчається студентами освітньо-кваліфікаційного рівня «бакалавр» спеціальності 123, «Інженерія програмного забезпечення» (галузь знань 12, «Інформаційні технології»). Дисципліна вивчається в першому та другому семестрах на 1 курсі.

Метою навчальної дисципліни «Програмування С++» є розвинення у студентів навичок складання програм мовою С++ для вирішення прикладних задач різного характеру.

Основними завданнями вивчення дисципліни «Програмування С++» формування у студентів теоретичні знань та практичних умінь застосування структурного та об'єктно-орієнтованого програмування мовою С++ та набуття наступних загальних та спеціальних компетентностей:

- Здатність до абстрактного мислення, аналізу та синтезу.
- Здатність застосовувати знання у практичних ситуаціях.
- Здатність ідентифікувати, класифікувати та формулювати вимоги до програмного забезпечення.
- Здатність розробляти архітектури, модулі та компоненти програмних систем.
- Володіння знаннями про інформаційні моделі даних, здатність створювати програмне забезпечення для зберігання, видобування та опрацювання даних.
- Здатність реалізовувати фази та ітерації життєвого циклу програмних систем та інформаційних технологій на основі відповідних моделей і підходів розробки програмного забезпечення.
- Здатність до алгоритмічного та логічного мислення.

Програмні результати навчання, яких студенти повинні досягти:

- Знати основні процеси, фази та ітерації життєвого циклу програмного забезпечення.
- Уміння вибирати та використовувати відповідну задачі методологію створення програмного забезпечення.
- Застосовувати на практиці ефективні підходи щодо проектування програмного забезпечення.
- Мотивовано обирати мови програмування та технології розробки для розв'язання завдань створення і супроводження програмного забезпечення.

Посібник включає теоретичний матеріал за ключовими темами дисципліни, наочні приклади і фрагменти програмного коду. Кожен розділ завершується

списком контрольних питань, які здобувачі можуть використати для самостійної роботи, а також для підготовки до поточного та підсумкового оцінювання.

В кінці посібника наведено перелік задач для самостійного опрацювання. Запропонований набір задач є авторською розробкою і представляє собою систему навчальних завдань, спрямованих на послідовне засвоєння базових і розширених понять процедурного програмування мовою С++. Задачі побудовані з урахуванням реальних предметних областей, що сприяє розвитку алгоритмічного мислення, умінь аналізу умов задачі, формалізації даних та побудови коректних програмних рішень. Задачі охоплюють різні рівні складності, що дозволяє реалізувати диференційований підхід до навчання.

1 ВЛАСТИВОСТІ ТА СПОСОБИ ПРЕДСТАВЛЕННЯ АЛГОРИТМІВ

1.1Визначення та властивості алгоритмів

Алгоритмом називається зрозуміле і точне розпорядження виконавцю виконати послідовність дій, спрямованих на досягнення зазначеної мети чи на розв'язання поставленої задачі.

В цьому означенні використовується поняття "виконавець". Що це означає? Під виконавцем алгоритму ми розуміємо будь-яку істоту (живу чи неживу), яка спроможна виконати алгоритм. Все залежить від того, якої мети ми намагаємося досягнути. Наприклад: риття ями (виконавці - людина або екскаватор), покупка деяких товарів (один із членів родини), розв'язування математичної задачі (учень або комп'ютер) тощо. Поняття алгоритму в інформатиці є фундаментальним, тобто таким, котре не визначається через інші ще більш прості поняття (для порівняння у фізиці - поняття простору і часу, у математиці - крапка).

Будь-який виконавець (і комп'ютер зокрема) може виконувати тільки обмежений набір операцій. Тому алгоритми повинні мати наступні властивості.

1. Зрозумілість. Щоб виконавець міг досягти поставленої перед ним мети, використовуючи даний алгоритм, він повинен уміти виконувати кожну його вказівку, тобто розуміти кожну з команд, що входять до алгоритму.

Наприклад: Мамі потрібно купити в магазині їжу. Виконавцем цього алгоритму може бути хтось із родини: батько, син, бабуся, маленька дочка тощо. Зрозуміло, що для тата достатньо сказати, які купити продукти, а далі деталізувати алгоритм не потрібно. Дорослому сину-підлітку необхідно детальніше пояснити в яких магазинах можна придбати потрібний товар, що можна купити замість відсутнього товару і таке інше. Маленькій дочці алгоритм необхідно деталізувати ще більше: де взяти сумку, щоб принести товар, яку решту грошей необхідно повернути з магазину, як дійти до магазину і як там поводитись (якщо дитина вперше йде за покупками).

2. Визначеність (однозначність). Зрозумілий алгоритм не повинен містити вказівки, зміст яких може сприйматися неоднозначно. Наприклад, вказівки "посоли за смаком", "прибери в квартирі" є неоднозначними, тому що в різних випадках можуть призвести до різних результатів. Крім того, в алгоритмах неприпустимі такі ситуації, коли після виконання чергового розпорядження алгоритму виконавцю не зрозуміло, що потрібно робити на наступному кроці. Наприклад: вас послали за яким-небудь товаром у магазин, та ще попередили "без (хліба, цукру і таке інше) не повертайся", а що робити, якщо товар відсутній?

Отож, *визначеність* - це властивість алгоритму, що полягає в тім, що алгоритм повинен бути однозначно витлумачений і на кожному кроці виконавець повинен знати, що йому робити далі.

3. Дискретність. Як було згадано вище, алгоритм задає повну послідовність дій, які необхідно виконувати для розв'язання задачі. При цьому, для виконання цих дій їх розбивають у визначеній послідовності на прості кроки. Виконати дії наступного розпорядження можна лише виконавши дії попереднього. Ця розбивка алгоритму на окремі елементарні дії (команди), що легко виконуються даним виконавцем, і називається дискретністю.

4. Масовість. Дуже важливо, щоб складений алгоритм забезпечував розв'язання не однієї окремої задачі, а міг виконувати розв'язання **широкого класу задач даного типу**. Наприклад, алгоритм покупки товару на деякому сайті інтернет-магазину буде завжди однаковий, незалежно від товару, що купується. Або алгоритм прання не залежить від назви білизни, що переться, і таке інше. Отож, під масовістю алгоритму мається на увазі можливість його застосування для вирішення великої кількості однотипних завдань.

5. Результативність. Взагалі кажучи, очевидно, що виконання будь-якого алгоритму повинне завершуватися одержанням кінцевих результатів. Тобто ситуації, що в деяких випадках можуть призвести до так званого "зациклення", повинні бути виключені при написанні алгоритму.

6. Правильність. При застосуванні алгоритму до допустимих вхідних даних має бути отриманий потрібний результат. Доведення правильності алгоритму – один з найскладніших етапів його створення. Найпоширеніша процедура перевірки правильності алгоритму – обґрунтування правомірності та перевірка правильності кожного з кроків при наборі тестів, підібраних так, щоб, наприклад, охопити всі допустимі класи вхідних і вихідних даних.

7. Ефективність **–** це здатність забезпечувати розв'язання задачі за мінімальний час з мінімальними затратами апаратних і програмних ресурсів. Для оцінки алгоритмів існує багато критеріїв. Найчастіше аналіз алгоритму (або, як кажуть, аналіз складності алгоритму) полягає в оцінці затрат часу на розв'язок задачі у розрахунку на одиницю вхідних даних. Фактично, ця оцінка зводиться до оцінки кількості базових елементарних операцій, на які можна розкласти даний алгоритм, оскільки кожна така операція виконується за конкретний, відомий відрізок часу. Ефективність алгоритму оцінюється також кількістю апаратних ресурсів, зокрема обсягом пам'яті, задіяної для виконання даного алгоритму.

1.2 Складність алгоритму

При оцінці ефективності алгоритму часто використовують поняття складності. Створення та реалізація алгоритму відповідно до свого призначення визначає його складність. Проте не існує інтегрованого показника складності алгоритму, хоча навіть існує спеціальний розділ інформатики – теорія алгоритмів,

що займається саме проблемами складності. Інтуїтивно можна виділити такі основні складові складності алгоритму:

1. Логічна складність - кількість людино-годин, витрачених на створення алгоритму.
2. Статична складність - довжина опису алгоритмів (кількість операторів).
3. Часова складність - час виконання алгоритму.
4. Ємкісна складність - кількість умовних одиниць пам'яті, необхідних для роботи алгоритму.
5. Функціональна складність – кількість ключових можливостей (задач), реалізованих в алгоритмі.

Головною метою теорії складності є забезпечення механізму класифікації алгоритмів за складністю. Складність алгоритму дозволяє визначитися з вибором ефективного алгоритму серед існуючих, що побудовані для розв’язання конкретної проблеми. А саме, вибір серед уже існуючих алгоритмів дозволяє не розглядати логічну та статичну складність, а оцінювати ті ресурси, що знадобляться під час реалізації обраних алгоритмів.

Визначення. Складність алгоритму – це кількісна характеристика, що відображує споживані алгоритмом ресурси під час свого виконання.

Основними ресурсами, що оцінюються, є час виконання і простір пам’яті.

Інтуїтивно це поняття досить зрозуміле. В алгоритмы є вхід - опис завдання, яке потрібно вирішити. На його розв’язання алгоритм витрачає певний час (тобто кількість операцій). Складність - це функція від довжини входу, значення якої дорівнює максимальній (за будь-якими входами даної довжини) кількості операцій, необхідних алгоритму для отримання відповіді.

Приклад 1.1. Нехай дана послідовність з нулів та одиниць і нам потрібно з'ясувати, чи є там хоч одна одиниця. Яку складність матиме алгоритм розв’язання цієї задачі?

Розв’язання. Нехай n – кількість символів в послідовності. Алгоритм буде послідовно перевіряти, чи немає одиниці в поточному місці заданої послідовності, а потім рухатися далі, поки вхід не скінчиться. Оскільки одиниця дійсно може бути тільки одна, для отримання точної відповіді на це питання в гіршому випадку доведеться перевірити всі n символів входу. В результаті отримуємо складність порядку cn, де c – кількість кроків, що потрібна для перевірки поточного символу і переходу до наступного. Оскільки такого роду константи сильно залежать від конкретної реалізації, математичного сенсу вони не мають, і їх зазвичай ховають за символом O (O-велике): в даному випадку фахівець із теорії складності визначив би, що алгоритм має складність

$$O(n);$$

іншими словами, він лінійний.

Існує і продовжує розширюватися клас варіантів поняття складності: складність знизу, зверху й у середньому, алгебраїчна складність, мультиплікативна складність, бітова складність, фундаментальні асимптотичні оцінки складності, оцінка алгоритмів за їх належністю до класів складності самих проблем, що вони розв'язують, і т. д.

Одним зі спрощених видів аналізу складності алгоритмів, що використовують при комп'ютерній їх реалізації, є асимптотичний аналіз алгоритмів. Він використовується з метою порівняння витрат часу та інших ресурсів різноманітними алгоритмами, призначеними для вирішення одного і того самого завдання. Досліджуючи зростання часу роботи алгоритму при вхідних даних досить великих розмірів, ми тим самим вивчаємо асимптотичну ефективність алгоритмів. Це означає, що нас цікавить тільки те, як час роботи алгоритму зростає зі збільшенням розміру вхідних даних, коли цей розмір збільшується до нескінченності. Зазвичай алгоритм, більш ефективний в асимптотичному сенсі, буде більш продуктивним для всіх вхідних даних, за винятком дуже маленьких. Нижче перелічені основні оцінки складності.

Позначення, що вводяться для опису асимптотичної поведінки часу роботи алгоритму, використовують функції, область визначення яких − множина невід'ємних цілих чисел N = {0,1,2, ...}. Подібні позначення зручні для опису часу роботи T(n) в найгіршому випадку як функції, визначеної тільки для цілих чисел, що становлять розмір вхідних даних.

Визначення. Функція складності алгоритму f(n) має оцінку Θ(тета) й записується як f(n) = Θ(g(n)), якщо існує невід'ємна функція g(n) та додатні n0, c1, c2 такі, що

$$c1g(n) \le f(n) \le c2g(n),$$

при $n > n0$.

У такому разі говорять, що функція g(n) є асимптотично точною оцінкою функції f(n), оскільки за визначенням функція f(n) не відрізняється від функції g(n) з точністю до сталого множника. Важливо розуміти, що Θ (g(n)) є не однією функцією, а множиною функцій для опису зростання f(n) з точністю до сталого множника. Іншими словами, функція f(n) належить множині Θ (g(n)), якщо існують додатні c1 та c2 , що дозволяють обмежити цю функцію у рамки між функціями c1g(n) та c2g(n) для достатньо великих значень n.

Наприклад, для методу сортування послідовності чисел алгоритмом heapsort оцінка трудомісткості становить $f(n) = \Theta (n\log_2 n)$, тобто в цьому разі $g(n) = n\log_2 n$.

З означення f(n) = Θ (g(n)) випливає, що g(n) = Θ (f(n)).

Інтуїтивно зрозуміло, що при асимптотичній точній оцінці асимптотично невід'ємних функцій, доданками нижчих порядків у них можна знехтувати,

оскільки при великих n вони стають неістотними. Навіть невеликої частки доданка найвищого порядку достатньо для того, щоб перевершити доданки нижчих порядків. Таким чином, для виконання нерівностей у визначенні Θ, достатньо як c1 вибрати значення, яке дещо менше коефіцієнта при самому старшому доданку, а як c2– значення, яке дещо більше цього коефіцієнта.

Тому коефіцієнт при старшому доданку можна не враховувати, тому що він лише змінює зазначені константи.

Приклад 2.2. Розглянемо квадратичну функцію $f(n) = an^2 + bn + c$, де a, b, c – константи, причому $a > 0$. Відкидаючи доданки нижчих порядків за перший та ігноруючи коефіцієнт a, одержимо $f(n) = \Theta(n^2)$.

Загалом кажучи, для будь-якого полінома p(n) маємо $p(n) = \Theta((n^d))$, де d – степінь полінома при $a^d > 0$. Оскільки будь-яка константа – це поліном нульового степеня, то сталу функцію можна записати як $\Theta(n^0)$ або $\Theta(1)$.

Визначення. Функція складності алгоритму f(n) має оцінку O(«О-велике») й записується як f(n) = O(g(n)), якщо існує невід'ємна функція g(n) та додатні n0, c такі, що

$$0 \le f(n) \le cg(n),$$

при $n > n0$.

Визначення. Функція складності алгоритму f(n) має оцінку Ω(«омега-велике») й записується як f(n) = Ω(g(n)), якщо існує невід'ємна функція g(n) та додатні n0 і c такі, що $0 \le, cg(n) \le f(n)$ при $n > n0$.

О–позначення застосовуються, коли необхідно вказати верхню межу функції з точністю до сталого множника. В позначеннях теорії множин $\Theta(g(n)) \subset O(g(n))$. Оскільки О-позначення описують верхню межу, то в ході їх використання для обмеження часу роботи алгоритму в найгіршому випадку ми отримуємо верхню межу цієї величини для будь-яких вхідних даних. Таким чином, асимптотична оцінка $O(n^2)$ для часу роботи алгоритму у найгіршому випадку застосовна для часу виконання завдання з будь-якими вхідними даними, чого не можна сказати про Θ-позначення. Наприклад, оцінка в $\Theta(n^2)$ для часу сортування вставками в найгіршому випадку не придатна для довільних вхідних даних. Наприклад, якщо вхідні елементи, що подаються на сортування, вже відсортовані в потрібному порядку, час роботи алгоритму сортування вставкою оцінюється як Θ(n).

Оскільки Ω - позначення використовуються для визначення нижньої межі часу роботи алгоритму в найкращому випадку, то вони визначають нижню межу часу роботи алгоритму при довільних вхідних даних. Наприклад, час роботи алгоритму сортування вставками знаходиться в межах між Ω(n) та O(n2), тобто між лінійною та квадратичною функціями від n.

Асимптотичний аналіз алгоритмів має не лише практичне, а й теоретичне значення. Так, наприклад, доведено, що всі алгоритми сортування, які ґрунтуються

на попарному порівнянні елементів, відсортують n елементів за час, не менший $\Omega(n\log_2 n)$.

1.3 Способи представлення алгоритмів

Алгоритм може задаватися різними способами:

– словесно, словами і реченнями природної мови – української, англійської тощо, в тому числі з використанням формул;

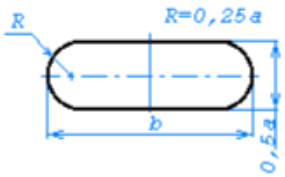
– графічно, за допомогою спеціальних схем із застосуванням умовних графічних позначень;

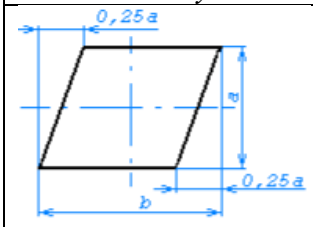
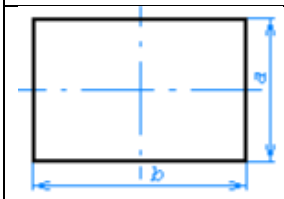
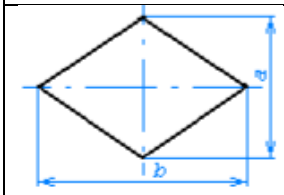
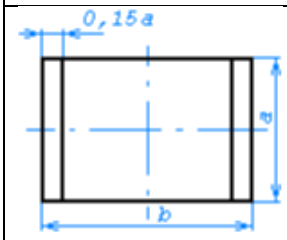
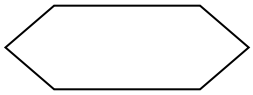
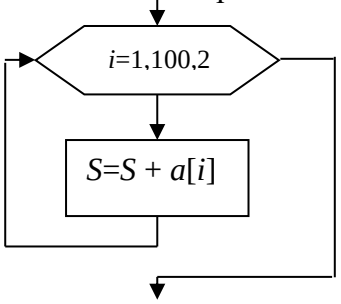
– символами спеціальної алгоритмічної мови.

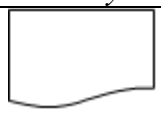
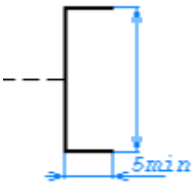
Запис алгоритму словами і реченнями природної мови найдоступніший для ширшого кола користувачів і не потребує спеціальної підготовки. Водночас така форма найменш точна та визначена, оскільки допускає різні тлумачення одного і того самого поняття внаслідок синонімічного багатства природних мов. Застосування спеціальних графічних схем для запису алгоритму дає змогу наочно бачити взаємозв'язок вхідних і вихідних величин на різних етапах виконання алгоритму. Крім того, алгоритм у такій формі доволі просто змінювати і коригувати.

В таблиці 1.1 наведені позначення стандарту, які використовуються для опису алгоритмів програм.

Таблиця 1.1 – Позначення елементів блок-схем (стандартні)

Графічне зображення блоку	*Найменування*	*Функція*
	Термінатор (пуск-зупинка)	Позначення початку і кінця алгоритму. При описі основного алгоритму він поміщається вгорі схеми і містить слово «Початок», а також розміщується в кінці алгоритму зі словом «Кінець» всередині. Так повинен починатися і закінчуватися кожен алгоритм. Блоки початку і кінця в алгоритмі на схемі тільки один раз. При описі допоміжних алгоритмів (підпрограм) їх схема також обов'язково містить блок початку і кінця в єдиному екземплярі. Але замість слова «початок» в блоці вказується ім'я алгоритму і змінні, значення яких передаються з викликає алгоритму. У блоці завершення замість слова «кінець» наводяться змінні, значення яких повертаються в викликає алгоритм по закінченню роботи допоміжного алгоритму.

Графічне зображення блоку	*Найменування*	*Функція*
	Дані (введення-виведення)	Перетворення даних у форму, придатну для обробки (введення) або відображення результатів обробки (виведення). Використовується для позначення будь-якої операції введення / виведення. В такому блоці зазначаються змінні, значення яких повинен ввести користувач в даному місці виконання алгоритму.
	Процес (операторний блок)	Виконання операції або групи операцій у результаті яких змінюється значення, форма представлення або розташування даних. Типове його використання – позначення оператора присвоювання. Операторний блок - це прямокутник, в який вписується деяка дія або вираз.
	Рішення (умовний блок)	В умовному операторі вказується умова (логічний вираз). З нього завжди виходить дві лінії, які позначені «+» і «-» (можлива позначка словами «так» і «ні»). Якщо умова істинна, то будуть виконуватися дії по стрілці, поміченої «+», інакше виконається оператор, до якого веде лінія, позначена «-».
	Зумовлений процес (підпрограма)	Відображає виконання процесу, який складається з однієї або кількох операцій, що визначені в іншому місці програми. Використовується для позначення виклику підпрограм (функцій). У блоці вказується ім'я допоміжного алгоритму (підпрограми) і значення змінних, які передаються в цей алгоритм. За типом і кількістю ці значення повинні збігатися зі змінними, вказаними в блоці початку допоміжного алгоритму.
	Цикл з відомим числом повторень	В такому блоці вказується змінна циклу, її початкове значення, кінцеве значення і величина кроку збільшення після кожного проходження циклу. Приклад схеми, що містить цикл з відомим числом повторень.  На схемі вказано цикл по знаходженню суми непарних елементів масиву. Стрілки позначають переходи між виконанням команд.
	Вивід на друк	В такому блоці вказується змінні або константи, значення яких виводиться на друк на даному етапі виконання алгоритму. Блок може

Графічне зображення блоку	*Найменування*	*Функція*
		використовуватися, як альтернатива блоку даних (виведення).
	З'єднувач	Відображає зв'язок між перерваними лініями потоку інформації на одній сторінці. При неможливості провести стрілку до блоку на одній сторінці (багато перетинів з іншими лініями) проводиться розрив лінії. Стрілка підводиться до даного блоку, в ньому вказується номер розриву. Потім в зручному для продовження місці вставляється даний блок з тим же номером і з нього триває розірвана стрілка. 
	Міжсторінковий з’єднувач	Застосовується аналогічно попередньому блоку, з тією відмінністю, що в ньому вказується сторінка, на яку виконується перехід, та, через крапку, номер переходу на цю сторінку. Наприклад 3.2 - другий перехід на третю сторінку.
	Коментар	Використовується для надання більш детальної інформації про кроки, процеси або групи процесів. Опис коментаря поміщається з боку квадратної дужки і охоплюється нею по всій висоті. Пунктирна лінія йде до описуваного елементу, або групи елементів. При цьому група виділяється замкнутою пунктирною лінією.
Розмір *a* повинен вибиратися з ряду 10, 15, 20 мм. Допускається збільшувати розмір a на число, кратне 5. Розмір b дорівнює 1,5a. При ручному виконанні схем алгоритмів допускається встановлювати b рівним 2a.		

Якщо операторний або умовний блоки мають більше одного входу, то зображення входів поєднується (рис. 1.1).

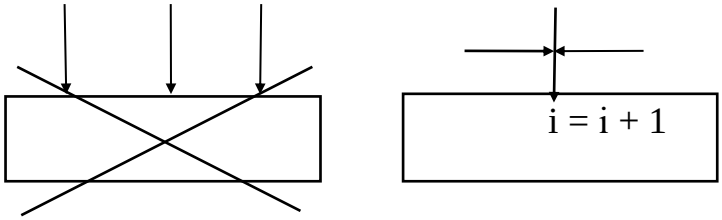


Рис.1.1 – Поєднання входів

Лінію потоку можна обривати, використовуючи на місці обриву *з'єднувачі*, якщо схему виконано на двох і більше аркушах або якщо символи, які з'єднуються,

розташовано на значній відстані один від одного. Якщо схема розривається, то використовується з'єднувач, в середині якого вказують номер блоку (сторінки), до якого (-ої) або від якого (-ої) здійснюється перехід. Відповідні символи з'єднання повинні мати одне (унікальне) позначення. Нумерувати блоки можна тільки ті, які пов'язані з передачею управління та з перевіркою умов.

Якщо розв'язання задачі складається з декількох алгоритмів, що реалізують окремі процедури або функції, то в кожному алгоритмі використовується власна нумерація переходів:

На зв'язках, що визначають послідовність виконання блоків, стрілки не обов'язкові, якщо їх напрям відповідає просуванню "зверху-вниз" і "зліва-направо" і, якщо вони не мають зламів. В інших випадках їх напрямок обов'язково позначають стрілкою. Лінію потоку, як правило, підводять до середини символу. Відстань між паралельними лініями потоку має бути не меншою 3 мм, між іншими символами — не меншою 5 мм.

<u>При складанні блок-схем необхідно керуватися наступними правилами:</u>

– блок-схема повинна містити **<u>одну початкову</u>**, **<u>одну кінцеву</u>** й **<u>кінцеве число інших вершин</u>**;

– входи й виходи різних вершин з'єднуються **<u>дугами</u>**, **<u>спрямованими</u>** тільки **<u>від виходу до входу</u>**;

– кожен **<u>вихід</u>** будь-якої вершини **<u>з'єднується тільки з одним входом</u>**;

– для будь-якої вершини існує, принаймні, один шлях із цієї вершини до кінцевої, що проходить через інші вершини у напрямку з'єднуючих їх дуг.

Блок-схемна форма представлення алгоритму найбільш поширена, бо вона наочна і дозволяє представляти алгоритм з різним ступенем деталізації. Перевагою блок-схем є те, що за їх допомогою можна наочно продемонструвати структуру алгоритму в цілому, відобразивши його логіку (показавши розгалуження шляхів розв'язання задачі залежно від виконання певної умови, численні повторення окремих етапів обчислювального процесу).

Контрольні питання

1. Дайте визначення алгоритму.
2. Які основні властивості повинен мати алгоритм?
3. У чому полягає скінченність алгоритму?
4. Що означає детермінованість алгоритму?
5. Чим алгоритм відрізняється від програми?
6. Наведіть приклади алгоритмів з повсякденного життя.

7. Яке значення має формалізація алгоритму?
8. Чому універсальність є важливою властивістю алгоритму?
9. Що розуміють під складністю алгоритму?
10. Від чого залежить ефективність алгоритму?
11. Яке значення має аналіз складності при розробці програм?
12. Як впливає розмір вхідних даних на складність алгоритму?
13. Наведіть приклади алгоритмів з різною складністю.
14. Чому важливо оцінювати складність до реалізації алгоритму?
15. Які існують способи подання алгоритмів?
16. У чому полягає сутність словесного опису алгоритму?
17. Які переваги та недоліки блок-схем?
18. Що таке псевдокод і для чого він використовується?
19. Коли доцільно використовувати графічне подання алгоритму?
20. Як програмний код пов’язаний з алгоритмом?
21. Які вимоги висуваються до коректного подання алгоритму?

2 ОСОБЛИВОСТІ ПОБУДОВИ ПРОГРАМИ С++

2.1 Ідентифікатори та ключові слова

Ідентифікатор - це послідовність символів, яка використовується для позначення імені змінної, структури, функції та інших елементів програми. Діапазон символів, які можна використовувати в ідентифікаторах в будь якому місці:

```
_ a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N
O P Q R S T U V W X Y Z
```

Цифри від 0 до 9 можна використовувати в ідентифікаторі на будь-якій позиції, крім першої.

Слід враховувати, що С ++ - чутлива до регістру мова, а це значить, що регістр символів має велике значення. Таким чином ідентифікатор *A* та ідентифікатор *a* будуть вважатися в програмі різними.

Ключові слова - це попередньо визначені зарезервовані ідентифікатори, які мають спеціальні значення. Їх використання в програмі в якості ідентифікаторів не допускається. До ключових слів відносяться:

- оператори;
- фундаментальні типи;
- інші зарезервовані ключові слова (повний перелік див. в довідковому матеріалі MSDN).

2.2Коментарі

Коментар - це текст, який призначений для програмістів і не обробляється компілятором. Зазвичай коментарі використовуються для створення заміток до коду для подальшого використання. Компілятор обробляє їх як пробіл.

Коментарі в С ++ записуються одним із таких способів:

1) Символи /* (коса риска і зірочка), за якими слідує будь-яка послідовність символів, включаючи переноси рядка, після чого ставляться символи */. Це той же синтаксис, який використовується в ANSI C.
2) Символи // (дві косі риски), за якими йде будь-яка послідовність символів. Символ переносу рядка, безпосередньо перед яким немає зворотної косої риски, завершує коментар, оформлений таким способом. Тому такі коментарі часто називають однорядковими.

Символи, які використовуються для оформлення коментарів (/*, */ і //), не мають спеціального значення всередині символьної константи, строкового літерала, або коментаря. Однак вкладення коментарів, оформлених першим способом, не допускається.

2.3Інструкції

Програма на C ++ складається з набору ***інструкцій***. Кожна інструкція (statement) виконує певну дію. В кінці інструкції в мові C ++ ставиться крапка з комою (;). Цей символ вказує на компілятору на завершення інструкції.

Наприклад, інструкція, яка об’являє змінну цілого типу:

```
int a;
```

Набір інструкцій може представляти ***блок коду***. Блок коду обмежується фігурними дужками, а інструкції розміщуються між відкриваючою та закриваючою фігурними дужками:

```
    {
        cin>>a;
        s=s+a;
    }
```

Блоки використовуються у випадках, коли в тіло оператора розгалуження, циклу чи тіло функції треба вставити більше однієї інструкції.

2.4Функція main

Кожна програма на мові C ++ повинна мати як мінімум одну функцію – функцію main (). Саме з цієї функції починається виконання програми: функція з ім'ям main - це початкова точка виконання для всіх програм на мовах C і C ++. Ім'я ***main*** фіксоване і не може використовуватися для ідентифікації інших елементів програми.

Функція є блоком коду, тому її тіло обрамляється фігурними дужками, між якими визначається набір інструкцій.

Синтаксис функції main виглядає наступним чином:

```
int main()
{

  return 0;
}
```

або, якщо потрібно,

```
int main(int argc, char *argv[], char *envp[])
{

  return 0;
}
```

Типи для параметрів argc і argv визначаються мовою. Імена argc, argv і envp є традиційними, але вони не є обов'язковими для компілятора.

Завершення роботи програми виконується функції main викликом оператора ***return*** із вказанням необхідного значення коду завершення.

До функції main застосовуються деякі обмеження, які не застосовуються до жодних інших функцій C ++. Функція main:

- не може бути перевантажена;
- не може бути оголошена як вбудована (inline);
- не може бути оголошена як статична (static);
- не можна отримати її адресу;
- викликати функцію неможливо.

2.5Директиви препроцесора

Препроцесор - це текстовий процесор, керуючий текстом файлу вихідного коду в ході першого етапу трансляції. Препроцесор не виконує синтаксичного аналізу тексту вихідного коду, але розбиває його на маркери для виявлення викликів макросів.

Директиви препроцесора зазвичай використовуються для того, щоб полегшити зміну вихідного коду програм і їх компіляцію в різних середовищах виконання, а також дозволяють препроцесору виконувати певні дії. Наприклад, препроцесор може вставляти вміст інших файлів в файл вихідного коду або відключати компіляцію частини файлу шляхом видалення розділів тексту.

Синтаксис директиви:

```
#директива
```

Знак решітки (#) повинен бути першим непробільним символом в рядку, що містить директиву; між знаком решітки і першою літерою директиви пробільні символи допускаються. Деякі директиви містять аргументи або значення.

Директиви препроцесора можуть перебувати в будь-якому місці файлу вихідного коду, але застосовуються лише до частини файлу, що слідує далі.

Перелік директив:

- #define
- #ifdef
- #undef
- #ifndef
- #error
- #import
- #if

- #else
- #elif
- #endif
- #include
- #using
- #line
- #pragma

Директива #include вказує препроцесору, що вміст заданого після директиви файлу необхідно обробити так, якби воно знаходилося у вихідній програмі в тій точці, в якій розташовується ця директива.

Приклади використання директиви:

```
#include <stdio.h>
#include "defs.h"
#include <iostream>
```

При виконанні директиви *#include ім'я файлу* замість неї підставляється весь вміст зазначеного файлу, що включається. Різниця між формою запису в лапках та в кутових дужками полягає в тому, в якому порядку препроцесор шукає файли заголовків, якщо шлях до них зазначений не повністю.

Якщо назва файлу вказана в лапках то порядок наступний:

1. У тому ж каталозі, де знаходиться файл з оператором #include.

2. У каталогах, відкритих в даний момент (в порядку, зворотному тому, в якому вони відкривалися). Пошук починається в каталозі батьківського include файлу, а потім виконується в каталогах всіх include файлів-прабатьків.

3. Шляхами, заданим усіма параметрами компілятора / I.

4. За шляхами, заданим в змінній середовища INCLUDE.

Якщо використовується форма з кутовими дужками, то препроцесор шукає файли, що включаються, в наступному порядку:

1. Шляхами, заданим усіма параметрами компілятора / I.

2. Якщо компіляція виконується з командного рядка - по шляхах, які задані в змінній середовища INCLUDE.

У файлах, що підключаються, зазвичай містяться визначення та функції, які будуть використовуватися у роботі програми.

2.6Приклад програми «Hello world»

Традиційна програма «Hello world» мовою C++ містить функцію main з інструкцією для виведення відповідного тексту і може виглядати наступним чином:

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

int main ()
```

```
{
  cout << "Hello world" << endl;
  system("pause");
  return 0;
}
```

Розглянемо детально кожен рядок програми

```
#include <iostream>
```

Директива препроцесора, яка підключає бібліотеку iostream. Ця бібліотека містить необхідні визначення, які потрібні для роботи з вхідними та вихідними потоками і використовуються для організації введення/виведення даних, в тому числі, при роботі з консоллю. Без підключення цієї бібліотеки неможливобуде використовувати оператори введення/виведення (>>, <<) та стандартні потоки cout, cin.

```
using namespace std;
```

Ця інструкція вказує компілятору, що надалі в програмі буде використовуватися ***стандартний простір імен std***.

Простір імен - це область, ***в рамках якої визначаються різні ідентифікатори*** (імена типів, функцій, змінних, і т. Д.). Простори назв використовуються для організації коду у вигляді логічних груп і з метою уникнення конфліктів імен, які можуть виникнути, особливо в таких випадках, коли база коду включає кілька бібліотек. Ідентифікатори ***за межами простору*** імен можуть отримати доступ до членів, використовуючи ***повне ім'я ідентифікатора***. Всі типи і функції стандартної бібліотеки C ++ оголошені в просторі імен std або в просторі імен, вкладеному в std.

При використанні інструкції *cout*<< без попередньої об'яви *using namespace std;* треба застосовувати формат запису *std::cout*<< .

```
int main ()
{
  cout << "Hello world" << endl;
  system("pause");
  return 0;
}
```

Блок основної програми, який власне, містить необхідні інструкції.

```
  cout << "Hello world" << endl;
```

Інструкція дозволяє вивести в консоль (потік cout) текстову константу "Hello world", а потім перевести курсор на наступний рядок (оператор << endl).

```
system("pause");
```

Інструкція містить виклик функції *system* з параметром "*pause*". Даний виклик дає команду операційній системі зробити паузу перед виконанням наступної інструкції. Для виходу зі стану паузи користувачеві треба натиснути будь-яку клавішу. Ця функція працює виключно під керуванням ОС Windows! При виконанні програми під ОС Linux по закінченню програми термінал не вимикається, тому можна побачити результати роботи програми без примусової паузи.

Альтернативою використання *system("pause")* є виклик функції, яка зчитує код символу, який натиснутий на клавіатурі: це функція *_getch();* Вона входить у бібліотеку <*conio.h*>, яку треба підключити за допомогою директиви *#include* перед тим як використовувати цю функцію. Ця функція працює у всіх операційних системах.

```
return 0;
```

Інструкція призводить до завершення програми із кодом 0, що відповідає успішному завершенню.

2.7Використання змінних в програмі

Будь-який алгоритм, реалізований на комп’ютері, призначений для обробки певної інформації. Ця інформація має зберігатися у пам’яті комп’ютера. Для обробки інформації в пам’яті комп’ютера було введено поняття змінної. Змінна в програмуванні – іменована частина (комірка) пам’яті комп’ютера, інформацію в якій можна змінювати. Основними операціями зі змінною є:

- запис у змінну нової інформації;
- читання інформації, що в ній записана.

Запис нової інформації у змінну здійснюється або за допомогою операції присвоювання, або із використанням оператора введення. Операції зміни знищують у змінній стару інформацію і записують нову.

Змінна в програмі має тип, ім'я та значення. Змінним варто давати осмислені імена, які будуть говорити про їх призначення. Тип змінної визначає, яку інформацію може зберігати змінна. Перед використанням будь-яку змінну треба визначити (об’явити). Синтаксис ***оголошення змінної*** виглядає наступним чином:

```
тип ідентифікатор;
```

Приклади об’явлених змінних:

```
int i; //змінна і цілого типу
float x; //змінна х дійсного типу
```

Конкретне значення об’явленій змінній можна задати декількома способами:

- за допомогою оператору введення (наприклад, *cin>>i;*);
- за допомогою оператору присвоювання (наприклад, *x=2.5;*);
- за допомогою ініціалізації.

Ініціалізація передбачає завдання значення безпосередньо при об’явленні змінної:

```
  int i=0; //змінна і цілого типу, яка має значення 0
  float x=2.0; //змінна х дійсного типу, яка має значення 2.0
```

Якщо змінну не ініціалізувати, то відбувається її ***ініціалізація за замовчуванням***. При оголошенні змінна отримує деяке значення, яке залежить від місця, де ця змінна визначена, а також від типу даних.

Важливо!

Перед використанням змінної краще явно призначати їй певне значення, а не покладатися на значення за замовчуванням.

Приклад програми, яка розраховує корінь лінійного рівняння ax+b=0:

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

int main ()
{
  float a, b; //коефіцієнти рівняння
  float x; //корінь рівняння
  cout << "Input a";
  cin >> a;
  cout << "Input b";
  cin >> b;
  x = -b/a;
  cout << “x=” << x;
  system("pause");
  return 0;
}
```

Контрольні питання

1. Що таке ідентифікатор у С++?
2. Які правила іменування ідентифікаторів?
3. Які символи дозволено використовувати в ідентифікаторах?
4. Що таке ключові слова?
5. Чому ключові слова не можна використовувати як ідентифікатори?
6. Навести приклади коректних і некоректних ідентифікаторів.
7. Яке значення має стиль іменування у програмі?
8. Для чого використовуються коментарі в програмі?

9. Які види коментарів існують у C++?
10. Чим відрізняються однорядкові та багаторядкові коментарі?
11. Чи впливають коментарі на виконання програми?
12. Які обмеження існують щодо вкладення коментарів?
13. Яку роль відіграють коментарі у навчальних програмах?
14. Які помилки пов’язані з неправильним використанням коментарів?
15. Чому коментарі важливі для супроводу коду?
16. Що таке інструкція в C++?
17. Яким символом завершується інструкція?
18. Що таке блок інструкцій?
19. Коли необхідно використовувати фігурні дужки?
20. Як блоки інструкцій застосовуються в умовних операторах?
21. Як блоки використовуються в циклах?
22. Які типові помилки пов’язані з інструкціями?
23. Чому структурованість інструкцій важлива?
24. Яке призначення функції main?
25. Чому кожна програма C++ повинна містити функцію main?
26. Який типовий синтаксис функції main?
27. Яке значення має оператор return?
28. Чи може програма містити кілька функцій main?
29. Які інструкції зазвичай розміщують у main?

3 ТИПИ ДАНИХ ТА ОПЕРАЦІЇ НАД НИМИ

3.1 Тип цілих чисел

Тип ціле число є основним для будь-якої мови програмування. Пов'язано це з тим, що вміст комірки пам'яті або регістра процесора можна розглядати як ціле число. Адреси елементів пам'яті також є цілі числа, з їх допомогою записуються машинні команди і т.д. Символи представляються в комп'ютері цілими числами - їх кодами в деякому кодуванні. Зображення також задаються масивами цілих чисел: для кожної точки кольорового зображення зберігаються інтенсивності її червоної, зеленої і синьої складової (в більшості випадків - в діапазоні від 0 до 255). Як кажуть математики, цілі числа дано зверху, все інше сконструювала з них людина.

Загальноприйнятий в програмуванні термін «ціле число» або «цілочисельна змінна», строго кажучи, не цілком коректний. Цілих чисел нескінченно багато, десяткова або двійковий запис цілого числа може бути дуже довгим і не поміщатися в області пам'яті, відведеної під одну змінну. Ціла змінна в комп'ютері може зберігати лише обмежену множину цілих чисел в деякому інтервалі. Якщо під цілочисельну змінну відводиться 4 байти, тобто 32 двійкових розряди, вона може зберігати числа від нуля до 2 в 32-му ступені мінус 1. Таким чином, максимальне ціле число, яке може зберігатися в цілочисельній змінній дорівнює:

$$2^{32} - 1 = 4294967295$$

Розряди (біти) цілого числа нумеруються від 0 до 31. Старший розряд з номером 31 зазвичай зберігає знак числа: 0 – позитивне число, 1 – негативне число. Якщо результат операції над цілими числами не вміщується в біти від 0 до 30, то може бути зачеплений знаковий розряд. Таким чином сума двох великих позитивних чисел може дати від'ємне число, або сума двох від’ємних чисел – позитивне число. Таку ситуацію називають **переповненням**.

Основні операції над значеннями цілого типу:

- додавання;
- віднімання;
- перемноження;
- ділення (результат – ціла частина);
- залишок від ділення.

Зверніть увагу!

Позначення операцій залежить від мови програмування. У мовах програмування сімейства С це відповідно +,-,,/,%.*

3.2 Тип дійсних чисел

До дійсних належать числа з дробовою частиною. Дійсні константи записуються в двох формах - з фіксованою десятковою крапкою або в експоненційному вигляді. У першому випадку точка використовується для поділу цілої та дробової частин константи. Як ціла, так і дрібна частини можуть бути відсутні.

Приклади: 1.2, 0.725, 1., .35, 0.

У трьох останніх випадках відсутня або дробова, або ціла частина. Десяткова точка повинна обов'язково бути присутньою, інакше константа вважається цілою. Відзначимо, що в програмуванні саме точка, а не кома, використовується для відділенні дробової частини.

Експоненціальна форма запису речової константи містить знак, мантису і десятковий порядок (експоненту).

Мантиса - це будь-яка позитивна матеріальна константа в формі з фіксованою точкою або ціла константа. Порядок вказує ступінь числа 10, на яку домножується мантиса. Порядок відділяється від мантиси буквою "e" (від слова exponent), вона може бути великою або малою. Порядок може мати знак плюс або мінус, в разі позитивного порядку знак плюс можна опускати.

Приклади:

1.5e + 6 константа еквівалентна 1500000.0

1e-4 константа еквівалентна 0.0001

-.75E3 константа еквівалентна -750.0

Дійсні числа представляються в пам’яті комп'ютеру в так званій експоненційній, або плаваючій, формі. **Дійсне число** r має вигляд

$$r = \pm 10^p * m$$

Представлення дійсного числа складається з трьох елементів:

1. Знак числа - плюс або мінус. Під знак числа відводиться один біт в двійковому поданні, він розташовується в старшому, тобто знаковому розряді. Одиниця відповідає знаку мінус, тобто негативного числа, нуль - знаку плюс. У нуля знаковий розряд також нульовий;

2. Показник ступеня *p*, його називають порядком. Порядок вказує ступінь десятки, на яку домножається число. Порядок може бути як позитивним, так і негативним (для чисел, менших одиниці). Під порядок відводиться фіксоване число двійкових розрядів, зазвичай вісім або одинадцять, розташованих в старшій частині двійкового представлення числа, відразу слідом за знаковим розрядом;

3. Мантиса m являє собою вміст значущих розрядів запису дійсного числа. Наприклад, для числа 1.2000700 мантиса буде дорівнювати 120007, а порядок буде дорівнювати -5.

У сімействі мови програмування С дійсним числам відповідають типи float і double. Елемент типу float займає 4 байта, в яких один біт відводиться під знак, вісім - під порядок, інші 23 - під мантису (насправді, в мантисі 24 розряду, але старший розряд завжди дорівнює одиниці, тому зберігати його не потрібно). Тип double займає 8 байтів, в них один розряд відводиться під знак, 11 - під порядок, інші 52 - під мантиссу. Насправді в мантисі 53 розряди, але старший завжди дорівнює одиниці і тому не зберігається. Оскільки порядок може бути позитивним і негативним, в двійковому коді він зберігається в зміщеному вигляді: до нього додається константа, яка дорівнює абсолютній величині максимального по модулю негативного порядку. У разі типу float вона дорівнює 127, в разі double - 1023. Таким чином, максимальний по модулю негативний порядок представляється нульовим кодом.

Основним типом є тип double, саме він найбільш природний для комп'ютера. У програмуванні слід по можливості уникати типу float, так як його точність недостатня, а процесор все одно при виконанні операцій перетворює його в тип double.

Операції над дійсними значеннями:

- додавання +
- віднімання -
- перемноження *
- ділення /

Зверніть увагу!

Є певні особливості виконання арифметичних операцій з дійсними числами.

При виконанні **додавання двох позитивних плаваючих чисел в двійковому представленні** відбуваються такі дії:

1. Вирівнювання порядків. Визначається число з меншим порядком. Потім послідовно його порядок збільшується на одиницю, а мантиса ділиться на 2, поки порядки двох чисел не зрівняються. Апаратно розподіл на 2 відповідає зрушенню двійкового коду мантиси вправо, так що ця операція виконується швидко. При зрушеннях праві розряди губляться, через це може відбутися втрата точності (в разі, коли праві розряди ненульові).

2. Складання мантис.

3. Нормалізація: якщо мантиса результату дорівнює або перевищила двійку, то порядок збільшується на одиницю, а мантиса ділиться на 2. В результаті цього

мантиса потрапляє в інтервал від 1 до 2. При цьому можлива втрата точності, а також переповнення, коли порядок перевищує максимально можливу величину.

Віднімання проводиться аналогічним чином. При множенні порядки складаються, а мантиси перемножуються як цілі числа, після чого у результату праві розряди відкидаються.

Дії з плаваючими числами через помилки округлення лише наближено відображають арифметику справжніх дійсних чисел. Так, якщо до великого дійсного числа додати дуже маленьке, то воно не зміниться. Дійсно, при вирівнюванні порядків все значущі біти мантиси меншого числа можуть вийти за межі розрядної сітки, в результаті чого воно стане рівним нулю. Наприклад, для типу double значення $1.0+10^{-16}$ буде дорівнювати 1.0. Це необхідно враховувати при організації процесу обчислень.

Крім втрати точності, при операціях з дійсними числами можуть відбуватися й інші неприємності:

1. Переповнення - коли порядок результату більше максимально можливого значення. Ця помилка часто виникає при множенні великих чисел.

2. Зникнення порядку - коли порядок результату негативний і дуже великий за абсолютною величиною, тобто порядок менше мінімально допустимого значення. Ця помилка може виникнути при діленні маленького числа на дуже велике або при множенні двох дуже маленьких за абсолютною величиною чисел.

Крім того, некоректною операцією є поділ на нуль. На відміну від операцій з цілими числами, переповнення і зникнення порядку вважаються помилковими ситуаціями і призводять до апаратного переривання роботи процесора. Програміст може задати реакцію на переривання - або аварійне завершення програми, або, наприклад, при переповненні привласнювати результату спеціальне значення плюс або мінус нескінченність, а при зникненні порядку - нуль.

Є спеціальна константа Not a Number, або NaN - двійковий код, який не є коректним поданням будь-якого дійсного числа.

Будь-які операції з константою NaN призводять до переривання, тому вона зручна при налагодженні програми - нею перед початком роботи програми ініціюються значення всіх дійсних змінних. Якщо в результаті помилки програміста при обчисленні виразу використовується змінна, якій не було присвоєно ніякого значення, то відбувається переривання через операцію зі значенням NaN і помилка швидко відстежується. На жаль, в разі цілих чисел такої константи немає: будь-який двійковий код представляє деяке ціле число.

3.3 Логічний тип

Логічний тип пов'язаний з результатами виконання операцій порівняння (відношень), до яких відносять:

– більше >
– більше або дорівнює >=
– менше <
– менше або дорівнює <=
– дорівнює ==
– не дорівнює !=

Зверніть увагу!

Вище наведені позначення прийняті в мовах програмування сімейства С, в інших мовах вони можуть бути відмінними.

Результат порівняння може бути істинним або хибним, відповідно true або false, які у пам'яті комп'ютера задаються значеннями 1 або 0.

Зі значеннями логічного типу можливе виконання специфічних логічних операцій, до яких зазвичай відносять:

– заперечення (ні)
– логічне «і» (кон'юнкція)
– логічне «або» (диз'юнкція)

В алгебрі логіки ці операції зазвичай позначається відповідно $\overline{A}$, A∧B (або крапка множення)), A∨B.

Їх таблиці значень мають вид:

A	$\overline{A}$
0	1
1	0

A	B	A∧B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A∨B
0	0	0
0	1	1
1	0	1
1	1	1

Всі інші логічні операції можуть бути реалізовані за допомогою наведених.

При розробці програм доцільно пам'ятати наступні логічні закони:

1. $p \vee 1 = 1$, $p \vee 0 = p$
2. $p \wedge 1 = p$, $p \cdot 0 = 0$
3. $p \wedge p = p$, $p \vee p = p$
4. $p \vee \bar{p} = 1$, $p \wedge \bar{p} = 0$
5. $p \vee q = q \vee p$, $p \wedge q = q \wedge p$
6. $p \wedge (q \vee r) = p \wedge q \vee p \wedge r$

   $p \vee q \wedge r = (p \vee q) \wedge (p \vee r)$
7. Закони де Моргана: $\overline{(p \vee q)} = \bar{p} \cdot \bar{q}$, $\overline{(p \cdot q)} = \bar{p} \vee \bar{q}$
8. $p \Rightarrow q = \bar{p} \vee q$

   $p \Leftrightarrow q = (p \Rightarrow q)(q \Rightarrow p)$

   $p \oplus q = \overline{p \Leftrightarrow q}$

   $p \downarrow q = \overline{(p \vee q)}$

   $p \mid q = \overline{(p \cdot q)}$
9. Закони поглинання:

   $p \cdot (p \vee q) = p$

   $p \vee p \cdot q = p$

   $p \vee \bar{p} \cdot q = p \vee q$
10. Закони склеювання:

$p \cdot q \vee \bar{p} \cdot q = q$, $(p \vee q)(\bar{p} \vee q) = q$
11. Закон подвійного заперечення: $\bar{\bar{p}} = p$.

В мовах програмування сімейства С

– заперечення позначається !

– логічне «і» позначається &&

– логічне «або» позначається ||

Логічні змінні та операції часто використовуються в умовних операторах та операторах циклів.

3.4 Символьний тип

Значенням символьної змінної є один символ з фіксованого набору. Такий набір зазвичай включає букви, цифри, розділові знаки, знаки математичних операцій і різні спеціальні символи (відсоток, амперсанд, зірочка, коса риска та ін.). Підкреслимо, що, на відміну від строкової змінної, символьна завжди містить рівно один символ (строкова містить рядок з декількох символів.)

Зверніть увагу!

В пам'яті комп'ютера ніяких символів не міститься. Символи представляються їх цілочисельними кодами в деякому фіксованому коді.

Кодування визначається трьома параметрами:

1. Діапазоном значень кодів. Наприклад, найпоширеніша в світі модель кодування ASCII (від слів american standard code of information interchange - американський стандартний код обміну інформацією) має діапазон значень кодів від 0 до 127, тобто вимагає сім біт на символ. Більшість сучасних кодувань мають діапазон кодів від 0 до 255, тобто один байт на символ. Нарешті, зараз у всьому світі здійснюється перехід на кодування unicode, яка використовує коди в діапазоні від 0 до 65535, тобто 2 байта на символ.

ASCII-таблиця

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

2. Множиною зображуваних символів. Наприклад, кодування ASCII містить букви латинського алфавіту, в західноєвропейському кодуванні до символів ASCII додані літери з Умлаут і акцентами, додаткові знаки пунктуації, зокрема, іспанські перевернуті знаки запитання й оклику, і інші символи європейських мов, заснованих на латинській графіці. Будь-яке з російських кодувань містить кирилицю.

3. Відображенням множини кодів на множину символів. Наприклад, російські кодування КОИ-8 (Код обміну інформацією восьмибітовий) і "Windows CP-1251", які традиційно використовуються в операційних системах Unix і MS Windows, мають один і той же діапазон кодів і один і той же набір символів, але відображення їх різні (одні і ті ж символи мають різні коди в кодуваннях ЯКІ-8 і Windows).

Існування різних кодувань букв кирилиці сильно ускладнює життя як програмістам, так і звичайним користувачам: файли при перенесенні з однієї системи в іншу доводиться перекодувати, періодично виникають труднощі при читанні листів, перегляді гіпертекстових сторінок і т.п.

З повсюдним переходом на кодування Unicode всі проблеми такого роду повинні зникнути. Кодування Unicode включає символи алфавітів всіх європейських країн і кирилицю. На жаль, більшість існуючих комп'ютерних програм пристосоване до подання одного символу у вигляді одного байта. Тому в даний час часто використовується проміжне рішення: комп'ютерні програми працюють з внутрішнім поданням символів в кодуванні Unicode (таке рішення прийнято в мовах Java і C #). При записи в файл символи Unicode приводяться до однобайтового коду відповідно до поточної мовної установки. При цьому, звичайно, частина символів втрачається - наприклад, в кодуванні Windows неможливо одночасно записати російські букви і німецькі Умлаут, оскільки Умлаут в західно-європейської кодуванні мають ті ж коди, що і російські літери в російському кодуванні.

Ще однією особливістю обробки символів є те, що символи, які мають однакове зображення, мають різні коди (наприклад, англійська A не дорівнює A в кирилиці), що часто призводить до різноманітних проблем.

3.5 Позначення типів даних в C++. Розміри типів

Позначення типів даних в C++ наведено в таблиці 3.1. Діапазони значень різних типів визначаються розміром комірки пам’яті, ку вони займають, та форматом зберігання (табл. 3.2).

Таблиця 3.1 – Типи даних C++

Категорія	Тип	Опис
Цілі числа	char	char - це цілочисельний тип, зазвичай містить члени основної таблиці кодування виконання (за замовчуванням в Microsoft C ++ це кодування ASCII). Компілятор C ++ обробляє змінні типу char, signed char і unsigned char як змінні різних типів. Змінні типу char підвищуються до типу int, як якщо б за замовчуванням вони мали тип signed char, якщо не використовується параметр компіляції /J. У цьому випадку вони розглядаються як тип unsigned char і підвищуються до типу int без розширення знака.
	bool	bool - це цілочисельний тип, який може мати одне з двох значень: true або false. Його розмір не визначений.

Категорія	Тип	Опис
	short	short int (або просто short) - це цілочисельний тип, розмір якого більше або дорівнює розміру типу char і менше або дорівнює розміру типу int. Об'єкти типу short можуть оголошуватися як об'єкти типу signed short або unsigned short.Signed short- синонім short.
	int	int - це цілочисельний тип, розмір якого більше або дорівнює розміру типу short int і менше або дорівнює розміру типу long. Об'єкти типу int можуть оголошуватися як об'єкти типу signed int або unsigned int.Signed int - синонім int.
	__int8, __int16, __int32, __int64, __int128	Ціле число із зазначенням розміру __int``n, де n - розмір в бітах цілочисельний змінної. (__int8, __int16, __int32, __int64 і __int128 - ключові слова для систем Microsoft. Доступність типів залежить від архітектури.)
	long	long (або long int) - це цілочисельний тип, розмір якого більше або дорівнює розміру типу int. Об'єкти типу long можуть оголошуватися як об'єкти типу signed long або unsigned long. Signed long - синонім long.
	long long	Більше, ніж unsigned long. Об'єкти типу long long можуть оголошуватися як об'єкти типу signed long long або unsigned long long. signed long long - синонім long long.
	wchar_t, __wchar_t	Змінна типу wchar_t позначає розширений символьний або мультибайтних символьний тип. За замовчуванням тип wchar_t є машинним типом, однак ви можете використовувати /Zc:wchar_t-, щоб зробити wchar_t визначенням типу для unsigned short .__wchar_t - синонім для машинного типу wchar_t для систем Microsoft. Щоб задати розширений символьний тип, перед символьним або строковим літералом слід використовувати префікс L.
З плаваючою комою	float	float - це тип з плаваючою комою найменшого розміру.
	double	double - це тип з плаваючою комою, розмір якого більше або дорівнює розміру типу float, але менше або дорівнює розміру типу long double.

Категорія	Тип	Опис
		Для систем Microsoft: представлення long double і double ідентично. Однак типи long double і double - це окремі типи.
	long double	long double - це тип з плаваючою комою, розмір якого більше або дорівнює розміру типу double (відмінності від типу double залежать від компілятора, зазвичай тип ідентичний до double)

Таблиця 3.2 – Розміри типів даних

Ім'я типу	Байти	Інші імена	Діапазон значень
int	4	signed	–2 147 483 648 to 2 147 483 647
unsigned int	4	unsigned	0 to 4 294 967 295
__int8	1	char(n)	–128 to 127
unsigned __int8	1	unsigned char	0 to 255
__int16	2	short, short int, signed short int	–32 768 to 32 767
unsigned __int16	2	unsigned short, unsigned short int	0 to 65 535
__int32	4	signed, signed int, int	–2 147 483 648 to 2 147 483 647
unsigned __int32	4	unsigned, unsigned int	0 to 4 294 967 295
__int64	8	long long, signed long long	–9 223 372 036 854 775 808 to 9 223 372 036 854 775 807
unsigned __int64	8	unsigned long long	0 to 18 446 744 073 709 551 615
bool	1	No	false or true
char(n)	1	No	–128 to 127 за умовчанням. При компіляції за допомогою / Y - від 0 до 255

Ім'я типу	Байти	Інші імена	Діапазон значень
signed char	1	No	–128 to 127
unsigned char	1	No	0 to 255
short	2	short int, signed short int	–32 768 to 32 767
unsigned short	2	unsigned short int	0 to 65 535
long	4	long int, signed long int	–2 147 483 648 to 2 147 483 647
unsigned long	4	unsigned long int	0 to 4 294 967 295
long long	8	none (but equivalent to __int64)	–9 223 372 036 854 775 808 to 9 223 372 036 854 775 807
unsigned long long	8	none (but equivalent to unsigned __int64)	0 to 18 446 744 073 709 551 615
float	4	No	3.4E +/- 38 (7 digits)
double	8	No	1.7E +/- 308 (15 digits)
long double	Same as double	No	Same as double

Контрольні питання

1. Що таке змінна?
2. Які правила оголошення змінних?
3. Що таке ініціалізація змінної?
4. Чим відрізняється ініціалізація від присвоєння?
5. Які значення мають неініціалізовані змінні?
6. Чому неініціалізовані змінні є небезпечними?
7. Які вимоги до імен змінних?
8. Як область видимості впливає на використання змінних?
9. Що таке тип даних?
10. Чому необхідно використовувати типи даних у програмі?
11. Які основні вбудовані типи даних існують у C++?
12. Чим відрізняються цілі та дійсні типи?

13. Яке призначення логічного типу?
14. Які арифметичні оператори підтримує C++?
15. Як працює з різними числовими типами оператора ділення?
16. Які логічні оператори використовуються в C++?
17. Що таке символьний тип?
18. Як тип даних впливає на обсяг пам’яті?
19. Які помилки пов’язані з неправильним вибором типу даних?

4 БАЗОВІ АЛГОРИТМІЧНІ СТРУКТУРИ ТА ОПЕРАТОРИ

Алгоритми можна представляти як деякі структури, що складаються із окремих базових (тобто основних) елементів – *базових алгоритмічних структур*. Кожен такий елемент відповідає певному типу алгоритмічного процесу.

Базові алгоритмічні структури:

- лінійна (слідування),
- розгалуження,
- циклу.

4.1 Лінійна структура

Лінійний алгоритм складається з однієї чи декількох дій (розпоряджень виконавцю), що повинні бути виконані у строгій послідовності, без всяких умов і в строгій відповідності з тим порядком, у якому записані оператори програми.

Типовим прикладом лінійного алгоритму є процедура обчислення за певними формулами.

Слід зауважити, що обчислення виразів є досить складним процесом, який потребує багато часу та спеціальної додаткової пам'яті для збереження проміжних результатів. Порядок обчислень у кожній мові визначається за пріоритетом операцій та скобками, використаними у записі виразу. Ці вирази можуть бути настільки складними, що можуть викликати збої в роботі програми. Крім того, довгі вирази погано читаються у коді програми. Тому доцільно розбивати складний вираз на декілька більш простих, пам'ятаючи, що запис виразу будь-якої складності повинен бути лінійним. Кожен вираз бажано розміщати тільки в одному рядку програми.

У якості прикладу лінійного алгоритму розглянемо пошук площі стін прямокутної кімнати:

1. Ввести довжину кімнати A.
2. Ввести ширину кімнати B.
3. Ввести висоту кімнати H.
4. Розрахувати площу стін S по формулі: S= 2*(A + B)*H.
5. Вивести значення змінної S як результат.

До лінійних операторів відносяться:

- присвоювання;
- введення;
- виведення;
- виклик допоміжного алгоритму.

Оператор присвоювання дозволяє змінити значення деякої змінної. Формат оператора:

<Ідентифікатор змінної> = <вираз>;

При виконанні оператора спочатку відбувається обчислення значення виразу, після чого в змінну заноситься результат обчислень.

Важливо!

Необхідно враховувати сумісність типів змінної і обчисленого виразу, а також розміри значень, які присвоюються.

В С++ оператор присвоєння має деякі різновиди, які дозволяють виконати в спрощеному вигляді присвоєння виразу, що містить су саму змінну, що стоїть зліва від оператора. Для базових математичних операцій оператор присвоєння модифікується наступним чином:

+ = Операція присвоювання-складання S+=a; замість S=S+a;

-= Операція присвоювання-віднімання S-=a; замість S=S-a;

= Операція присвоювання-множення S=a; замість S=S*a;

/ = Операція присвоювання-ділення S/=a; замість S=S/a;

% = Операція присвоювання-залишку від ділення S%=a; замість S=S%a;

Такі самі конструкції оператора можуть застосовуватись до логічних операцій.

Оператор введення призначений для занесення в змінні з зовнішнього пристрою (наприклад, з клавіатури) деяких значень для їх подальшої обробки.

Важливо!

Список введення може містити тільки змінні.

В мовах програмування високого рівня немає окремого фіксованих операторів для реалізації операцій введення/виведення!

Оператор виведення призначений для відображення даних з пам'яті комп'ютера на зовнішній пристрій (наприклад, екран, принтер і ін.).

Рекомендується виводити:

- вхідні дані (для контролю правильності введення),
- проміжні дані (для контролю ходу виконання завдання),
- вихідні дані (виводяться обов'язково, в тому числі, пояснення до них).

Для поліпшення «читабельності» алгоритму та показників юзабіліті програми бажано виводити текстові пояснення до даних (наприклад, "Обчислення суми елементів рядка", "Знаходження найбільшого елемента в стовпці", "Результати роботи алгоритму:", "Таблиця результатів:" і т.д.).

Операції введення/виведення в С++ реалізовано декількома способами, і для спрощення сприйняття на перших етапах написання програм рекомендується

використовувати потокове введення/виведення. Детальніше про потоки описано в розділі 5.

```
#include <iostream>
using namespace std;
int main() {
int x,y;
cout << " input 2 number: ";
cout << " x:\t ";
cin >> x;
cout << " y:\t ";
cin>> y;
cout << "Rezult:"<<endl<< x << ' * ' << y << ' = ' << x*y;
}
```

4.2 Розгалужені алгоритми

У випадках, коли перетворення інформації може здійснюватися за різними схемами, залежно від властивостей вхідних даних або проміжних результатів, використовуються *розгалужені алгоритми*. В таких алгоритмах передбачаються всі можливі варіанти обробки інформації, кожний з яких розробляється як окрема гілка алгоритму, а вибір однієї з них для виконання здійснюється за допомогою перевірки деякої умови, що відображає властивості інформації, яка використовується у процесі перетворення.

Для представлення таких алгоритмів використовуються алгоритмічні конструкції *вибору* (*розгалуження*). Дана алгоритмічна структура в залежності від результату перевірки певної умови здійснює вибір одного з альтернативних шляхів роботи алгоритму. Кожен із шляхів веде до спільного виходу, так що робота алгоритму триватиме незалежно від того, який шлях буде обраний.

Структура розгалуження існує в чотирьох основних варіантах:

- «якщо-то-інакше» (повне розгалуження);
- «якщо-то» (неповне розгалуження);
- «вибір-інакше» (повний багатоваріантний вибір);
- «вибір» (неповний багатоваріантний вибір).

Повне розгалуження передбачає виконання дій і у разі виконання, і у разі невиконання заданої умови. Графічне представленням такої структури у блок-схемах має вид, наведений на рисунку 4.1.

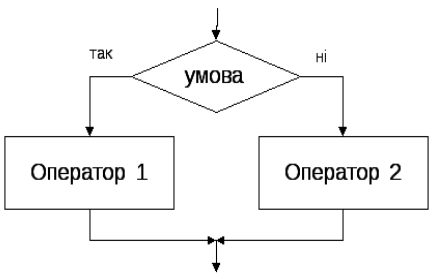


Рис. 4.1 - Загальний вид оператора розгалуження

При цьому умова формулюється таким чином, щоб відповідь перевірки була «так» чи «ні». Як правило, при запису умови використовуються операції порівняння та логічні операції. Для спрощення запису умов доцільно використовувати формули алгебри логіки.

Зазначимо, що у якості оператору 1 чи оператору 2 може виступати не одна дія, а кілька, як представлено на рисунку 4.2.

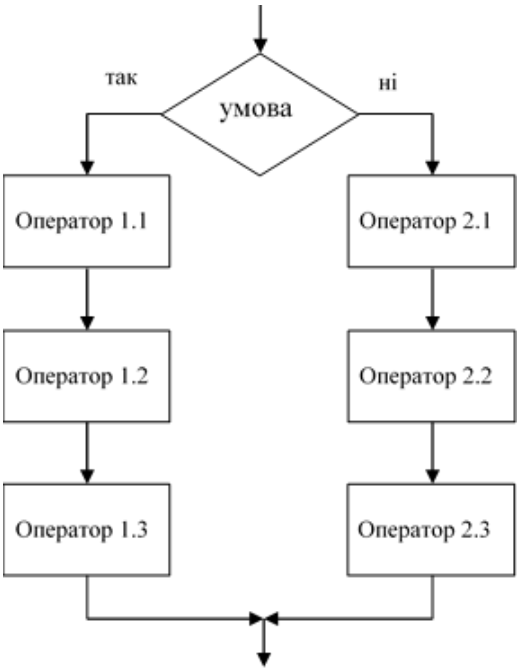


Рис. 4.2. – Блок-схема повного розгалуження

У мові C++ для організації розгалуження використовується оператор ***if***. Конструкція ***else*** задає альтернативну гілку виконання для випадку, коли умова не виконується.

Загальна форма оператору повного розгалуження в C++:

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

Умовою в if може бути будь-який вираз, що приводиться до логічного типу bool, зокрема:

- порівняння (==, !=, <, >, <=, >=);
- логічні операції (&&, ||, !);
- значення змінних або результат функцій.

Неповне рогалуження передбачає виконання дій тільки у разі виконання, або у разі невиконання заданої умови. Тобто, одна із її гілок взагалі не передбачає

ніяких дій. Графічне представленням таких структур у блок-схемах має наступний вид, представлений на рисунку 4.3.

Рис. 4.3 – Блок-схема неповного розгалуження

Зверніть увагу!

У випадку, коли має бути виконаний оператор з гілки «ні», до умови застосовується логічна операція «ні».

Залежно від того, на скільки гілок розгалужується алгоритм, він може бути простим або складним. Для простого розгалуженого процесу перевіряється одна умова, для складного – дві чи більше умов, кожна з яких відокремлює одну гілку (див. рис. 4.4).

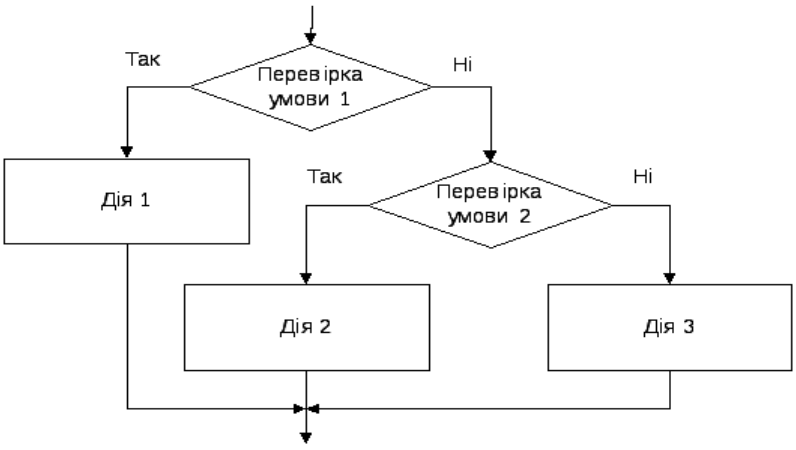


Рис. 4.4 – Блок-схема складного каскадного розгалуження

Складне розгалуження в більшості випадків реалізується в мовах програмування вкладенням умовних операторів. Якщо присутні декілька каскадних умов, то вони записуються наступним чином:

```
if (умова1) {
   // ...
} else if (умова2) {
   // ...
} else if (умова3) {
   // ...
} else {
   // ...
}
```

Перевірка умов відбувається послідовно зверху вниз. Виконується перша гілка, умова якої є істинною. Якщо жодна умова не істинна, виконується блок else (за наявності).

Особливості використання оператору if в C++:

- якщо тіло if або else містить одну інструкцію, фігурні дужки можна опустити, але це не рекомендується з точки зору читабельності;
- оператор if може бути вкладеним у інший if;
- умови обчислюються лише до моменту визначення гілки виконання (коротке замикання для && та ||).

Для спрощених перевірок в C++ часто використовують ***тернарний оператор***. Умовний (тернарний) оператор '?:' використовується зазвичай у тих випадках, якщо умова і код, який треба виконати в результаті перевірки умови, дуже прості. Наприклад, запитати в користувача, чи хоче він продовжити працювати в програмі, чи хоче вийти з неї. Оператор '?:' у мові C++ - це скорочена форма оператора if … else, яка дозволяє вибрати одне з двох значень залежно від умови. Загальна форма оператора:

умова ? вираз_якщо_істина : вираз_якщо_хиба;

Читається так: «якщо умова істинна - використати перший вираз, інакше - другий». Принцип роботи:

1. Обчислюється умова.
2. Якщо вона істинна (true) — виконується вираз після ?.
3. Якщо хибна (false) — виконується вираз після :.
4. Результатом усього оператора є значення вибраного виразу.

Приклад 4.1.

```
int x = 5;
int result = (x > 0) ? 1 : -1;
```

В цьому прикладі спочатку перевіряється умова x > 0. Якщо x більше нуля – result отримає значення 1, якщо ні – result отримає значення -1.

Для x = 5 результат буде: result = 1;

Те саме з використанням оператору if буде виглядати наступним чином:

```
int result;
int x = 5;

if (x > 0) {
    result = 1;
} else {
    result = -1;
}
```

Збільшення кількості умов робить алгоритм більш заплутаним, він втрачає наочність, перевірити його правильність досить складно. У таких випадках необхідно перехід до будь-якої гілки розгалуженого алгоритму пов’язати з деякою змінною, кожне значення якої відповідатиме одній із гілок розгалуження, тобто одному з варіантів обробки інформації. Тоді всі логічні блоки алгоритму об’єднуються в один блок аналізу цієї змінної, який матиме не два виходи, а стільки, скільки існує варіантів обробки. Таке розгалуження називають *багатоваріантним* (див. рис 4.5). В C++ такій конструкції відповідає оператор вибору (switch). Оператор ***switch*** є зручнішою альтернативою довгому ланцюжку if … else if … else, коли порівняння відбувається з конкретними значеннями.

Рис. 4.5 – Блок-схема багатоваріантного вибору

Загальна форма оператору switch:

```
switch (вираз) {
    case значення1:
        // інструкції
        break;

    case значення2:
        // інструкції
        break;

    // ...

    default:
        // інструкції, якщо жоден варіант не підійшов
}
```

Принцип роботи:

1. Обчислюється вираз у дужках switch.
2. Отримане значення порівнюється з кожним case.
3. Якщо знайдено відповідний case, виконання починається з цього місця.
4. Виконання триває, доки не буде виконано break або не закінчиться switch.
5. Якщо жоден case не збігся — виконується блок default (за наявності).

Приклад 4.2.

```
int day = 3;

switch (day) {
    case 1:
        cout << "Понеділок";
        break;
    case 2:
        cout << "Вівторок";
        break;
    case 3:
        cout << "Середа";
        break;
    case 4:
        cout << "Четвер";
        break;
    case 5:
        cout << "П’ятниця";
        break;
    default:
        cout << "Невідомий день";
}
```

break перериває виконання оператора switch. Ситуація, коли break відстуні, називається «провалюванням». Іноді воно використовується свідомо, але для початківців є типовою помилкою.

Приклад 4.3.

```
int x = 1;

switch (x) {
    case 1:
        cout << "Один ";
    case 2:
        cout << "Два ";
    case 3:
        cout << "Три ";
}
```

Результатом роботи буде

Один Два Три

Приклад 4.4. Свідоме використання провалювання. Тут кілька case ведуть до одного блоку коду.

```
char grade = 'B';

switch (grade) {
    case 'A':
    case 'B':
    case 'C':
        cout << "Зараховано";
        break;
    case 'D':
    case 'F':
        cout << "Не зараховано";
        break;
}
```

У switch можна використовувати:

- цілі типи (int, char, short, long);
- тип enum;
- починаючи з C++17 — тип std::string_view не підтримується, лише цілі значення.

Не можна використовувати:

- float, double;
- діапазони значень;
- логічні вирази (x > 5).

Блок default можна розташувати в будь-якому місці блоку switch(). Наприклад між першим і другим case . Його код у будь-якому разі виконається тільки тоді, якщо не знайдеться потрібного значення в блоках case. Але доцільно розташовувати його саме наприкінці, як роблять більшість програмістів, це таке негласне правило. default не є обов'язковим, його в switch() може і не бути зовсім. У такому разі, якщо жодне значення блоків case не збігається з тим, що прийняв switch(), програма просто перейде на наступний рядок коду, розташований під switch().

4.3 Цикли

У деяких алгоритмах передбачається можливість багаторазового виконання деякої сукупності дій. Такі алгоритми називають *циклічними* (циклом), а їх повторювану частину - *тілом циклу*.

Для побудови циклічного алгоритму необхідно:

- визначити всі дії, які необхідно виконати до входу в цикл, тобто провести підготовку циклу;
- визначити всі операції, які ввійдуть до циклу;
- скласти умову повторення виконання операцій циклу або виходу з циклу.

У циклічних алгоритмах обчислення проводяться до тих пір, поки не виконається (або не порушиться) вказана в заголовку циклу умова. Залежно від місця перевірки умови, команди повторення можна розділити на дві групи: «цикл з передумовою» і «цикл з постумовою». Для представлення циклічних алгоритмів використовуються конструкції повторення, які реалізуються одним із наведених далі способів.

4.3.1 Цикли з передумовою (while, for)

Цикл з передумовою буде виконуватися до тих пір, поки логічний вираз в заголовку циклу є істинною.

Для правильної організації роботи циклу з передумовою необхідно:

- до перевірки логічного виразу в заголовку циклу ініціювати параметри циклу (*задати їм початкове значення, наприклад, i: =* $i_{поч.}$);
- правильно скласти логічний вираз, що визначає умову входу в цикл (*наприклад, i <=* $i_{кін.}$);
- передбачити зміну значення параметра циклу (*наприклад, i = i + h, де h - величина кроку зміни параметра циклу*).

До циклів з передумовою відносяться:

- цикл while;
- цикл for.

Блок схема виконання циклу ***while*** наведена на рис.4.6.

Структура циклу ***while*** в мові С++:

```
Оператори_підготовки_циклу;
while (логічний вираз)
{
Оператори_тіла_циклу;
Оператори_переходу_до_наступної_ітерації_циклу;
}
```

Послідовність операторів тіла циклу виконується до тих пір, поки логічний вираз повертає значення true. Як тільки вираз стає рівним false, виконання циклу while припиняється і управління передається наступному за циклом while оператору.

Підготовка циклу: всім величинам, що змінюються по рекурентним формулами, присвоюються початкові значення (в тому числі – змінним в заголовку циклу)

Заголовок циклу: перевірка умови продовження циклу

Тіло циклу:

- реалізація необхідних в завданні дій;
- отримання нових значень величин, які використовуються в заголовку циклу.

Рис.4.6 – Блок схема циклу while

Приклад 4.5.

Дано дійсне позитивне число *a*. Знайти таке найменше *n*, при якому $1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n}>a$.

Рішення задачі: На початку значення суми є меншим за значення *a*. При проходженні кожної ітерації значення суми поступово зростає. В якийсь момент (при якомусь значенні *n*) ця сума стане вищою за значення *a*. Цей момент (значення *n*) потрібно зафіксувати.

Для обчислення суми та кількості кроків використовуємо метод накопичення.

Фрагмент коду, що розв‘язує дану задачу:

```
float a;
int n;
float s;

//підготовка параметрів циклу та змінних для обчислення

cout<<”a=”;
cin>>a;

s = 0.0;

n = 0;

while (s <= a) //заголовок циклу: продовжуємо, поки сума <= значення a
{
    s = s + 1.0/n; //обчислення наступного значення суми
    n++;               //перехід до наступного елементу ряду

}
```

Цикл ***for*** також відноситься до циклів з передумовою. Його особливість полягає в тому, що присвоювання початкового значення параметра циклу (*наприклад, $i = i_{поч}$.*), що змінюється по рекурентним формулам, перевірка умови продовження циклу (*наприклад, $i \le i_{кін}$.*) і отримання нового значення параметру циклу ($i = i + крок$), виконується в заголовку циклу. Блок схема виконання циклу ***for*** наведена на рис.4.7.

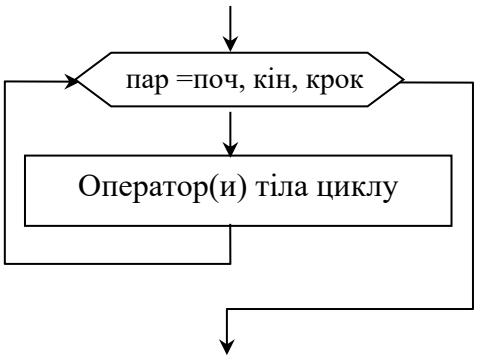


Рис.4.7 – Блок схема циклу for

В мові С++ цикл ***for*** може мати дуже широку реалізацію та застосування.

Загальна форма оператора циклу ***for***:

for*(ініціалізація; логічний_вираз; підготовка_наступної_ітерації)*
{
Оператори_тіла_циклу;
}

Зверніть увагу!

Блоки в заголовку циклу for відокремлюються як і оператори мови – за допомогою ;

Перший і третій блоки можуть містити декілька операторів, блок логічного виразу – тільки один вираз. Якщо перший чи третій блоки містять декілька операторів, то вони відокремлюються один від одного комою.

Блок ініціалізації. Містить оператори присвоювання, в яких встановлюються початкові значення змінних циклу та початкові значення змінних, які використовуються в тілі циклу для вирішення задачі.

Блок логічного виразу. Визначає можливість подальшого виконання циклу. Послідовність операторів тіла циклу виконується до тих пір, поки логічний вираз повертає значення true. Як тільки вираз стає рівним false, виконання циклу ***for*** припиняється і управління передається наступному за циклом ***for*** оператору.

Блок підготовки наступної ітерації. Зазвичай визначає, як будуть змінюватись значення змінних циклу після кожної ітерації. Оператори цього блоку виконуються **<u>після</u>** кожного виконання операторів тіла циклу!

Фрагмент коду, що розв‘язує задачу з прикладу 4.5:

```
float a;
int n;
float s;

cout<<”a=”;
cin>>a;

n = 0;

//підготовка та зміна параметрів циклу виконується безпосередньо в циклі

for (s = 0.0; s <= a; s = s + 1.0/n)
{
    n++;

}
```

Зверніть увагу!

Одне й те ж рішення може бути записано за допомогою циклу for декількома способами, які працюють абсолютно однаково!

Наприклад, фрагменти 1, 2 і 3 виконують одне й те ж!

```
//Фрагмент 1
float a;
int n;
float s;

cout<<”a=”;
cin>>a;

n = 0;

for (s = 0.0; s <= a; s = s + 1.0/n)
    n++;
```

```
//Фрагмент 2
float a;
int n;
float s;

cout<<”a=”;
cin>>a;

s = 0.0;
n = 0;
for (; s <= a;)
{
        n++;
s = s + 1.0/n;
}
```

```
//Фрагмент 3
```

```
float a;
int n;
float s;

cout<<”a=”;
cin>>a;
for (s = 0.0, n = 0; s <= a; n++, s = s + 1.0/n);
```

Фрагмент 2 специфічний тим, що в заголовку циклу залишився тільки логічний вираз. Якщо ви стикаєтеся із такою ситуацією, то краще оформити такий цикл за допомогою оператору ***while***.

Фрагмент 3 специфічний тим, що в результаті перенесення операторів присвоєння, що відповідають за ініціалізацію, перехід до наступної ітерації та обчислення самої задачі, **тіло циклу стало пустим!** Таких ситуацій слід уникати, оскільки **це значно знижує розуміння коду**. Крім того, помилки в записі операторів в третьому блоці може призвести до обчислення зовсім інших результатів. Наступні фрагменти **дуже схожі, але призводять до різних результатів**!

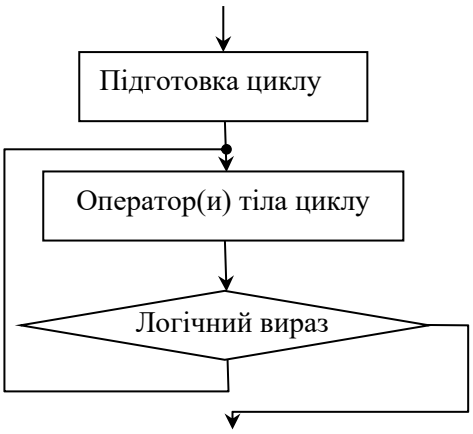
```
//Фрагмент 3
float a;
int n;
float s;

cout<<”a=”;
cin>>a;
for (s = 0.0, n = 0; s <= a; n++, s = s + 1.0/n);

//Фрагмент 4
float a;
int n;
float s;

cout<<”a=”;
cin>>a;
for (s = 0.0, n = 0; s <= a; s = s + 1.0/n, n++);
```

4.3.2 Цикл з післяумовою. (do … while)

Цикл ***do … while*** відноситься до категорії циклів з післяумовою. Його доцільно використовувати у випадках, коли ітерацію потрібно зробити хоча б 1 раз. На відміну від циклів for та while, у циклі ***do…while*** умова перевіряється при виході з циклу (а не при вході в цикл). Блок схема виконання циклу ***do…while*** наведена на рис.4.8.

Загальна форма оператора циклу do…while в мові С++:

```
do
{
   Оператори_тіла_циклу;
}
 while (логічний_вираз);
```



Всім величинам, що змінюються по рекурентним формулами, присвоюються початкове значення.

Реалізація необхідних в завданні дій. Отримання нових значень величин, які використовуються в логічному виразі.

Перевірка умови продовження циклу.

Рис.4.8 – Блок схема циклу do…while

Приклад 4.6

Наведений фрагмент дозволяє організувати перевірку введення додатного числа N. Введення виконується до тих пір, поки користувач не введе додатне число.

```
int N;
do {
    cout<<”Enter positive N”;
    cin>>N;
    if(N <=0)
       cout<<”N is not positive!”;
  } while (N <=0);
```

З алгоритмічних структур зазначених типів можна будувати складні циклічні процеси із ***вкладеними циклами***. У цьому випадку виділяються *внутрішній* і *зовнішній* цикли. Для кожної зміни значення параметра у зовнішньому циклі відбувається багаторазове виконання дій у внутрішньому циклі, який називається *вкладеним*. Кількість вкладених циклів не обмежується.

Треба зауважити, що внутрішній і зовнішній цикли можуть бути різних типів, кожен, у свою чергу, бути складним. Але вони повинні цілком вкладатися один в один. При цьому важливо пам’ятати про правильну підготовку до початку кожного з циклів.

Комбінуючи базові алгоритмічні структури між собою, можна будувати алгоритми будь-якої складності. Цих структур достатньо для створення найскладнішого алгоритму.

4.4 Рекурентні обчислення

Рекурентні обчислення отримали широке розповсюдження в програмуванні, особливо при організації циклів. Будемо розуміти під рекурентним обчисленням ситуацію, коли наступне значення змінної розраховується з використанням її попереднього значення.

Наприклад, для розрахунку нового значення деякої суми S використовується її попереднє значення (значення на попередньому кроці).

Для програмування рекурентних обчислень спочатку необхідно розробити їх математичну модель у вигляді рекурентної формули.

У загальному випадку рекурентним співвідношенням називається формула виду:

$$a_{n+1}=F(a_n, a_{n-1},\dots,a_{n-k+1}),$$

де F - деяка функція від k аргументів, яка дозволяє обчислити наступні члени числової послідовності через значення попередніх членів.

Приклади рекурентних формул:

1. Рекурентне співвідношення арифметичної прогресії:
   $a_{n+1}= a_n + d.$
2. Рекурентне співвідношення геометричної прогресії:
   $a_{n+1}= a_n \cdot q.$
3. Послідовність Фібоначчі:
   $a_{n+1}= a_n + a_{n-1},\ a_1=1,\ a_2=1.$

У найбільшій кількості випадків рекурентна формула має вигляд:

$$a_{n+1} = a_n + r(n) \text{ або } a_{n+1} = a_n * r(n)$$

де n – номер значення змінної, що розраховується, $r(n)$- вираз, що показує зв'язок між теперішнім та майбутнім значеннями змінної.

Для початку обчислень має бути задане початкове значення a_0.

Основна задача при розробці рекурентної формули – це пошук виразу $r(n)$. Якщо відомий математичний вираз для розрахунку елемента послідовності $a_n = g(n)$, то для пошуку $r(n)$ можна використати (в залежності від типу формули) наступний підхід:

$$r(n) = a_{n+1} - a_n = g(n+1) - g(n) \text{ або } r(n) = a_{n+1}/a_n = g(n+1)/g(n)$$

Наприклад, розглянемо пошук рекурентної формули для розрахунку n!.

Як відомо, $n!= 1 * 2 * 3 * \dots * n,\ \ 0!=1.$

$$r(n) = \frac{a_{n+1}}{a_n} = \frac{(n+1)!}{n!} = \frac{1 * 2 * 3 * \dots * n * (n+1)}{1 * 2 * 3 * \dots * n} = n + 1$$

Таким чином, для розрахунку $n!$ можна використовувати рекурентну формулу:

$$a_{n+1} = a_n * (n+1), \quad a_0 = 1$$

Наголосимо, що рекурентні обчислення використовуються лише там, де це необхідно. Як правило, вони застосовуються в задачах пошуку сум або добутків елементів послідовностей, в розрахунках формул зі ступенями або факторіалами.

У складних випадках в тілі циклу може бути використано кілька рекурентних формул, які необхідно правильно узгодити між собою.

Приклад 4.7. Знайти суму значень 1!,2!,3!,4!,5!,6!,7!,8!,9!,10!

Рішення. Формально цю задачу можна представити, як пошук суми значень елементів послідовності $a_n = n!$ для $n = \overrightarrow{1,10}$.

Цю задачу можна також записати у вигляді:

$$S = \sum_{n=1}^{10} n!$$

Очевидно, що для розрахунку значення суми необхідно використати рекурентну формулу:

$$S = S + n!$$

Однак для розрахунку $n!$ доцільно також використати рекурентну формулу. Введемо змінну *nf* для позначення $n!$. Для її розрахунку будемо використовувати раніше знайдену рекурентну формулу факторіалу:

$$nf = nf * (n + 1).$$

Алгоритм розрахунку має вид, представлений на рис. 4.9.

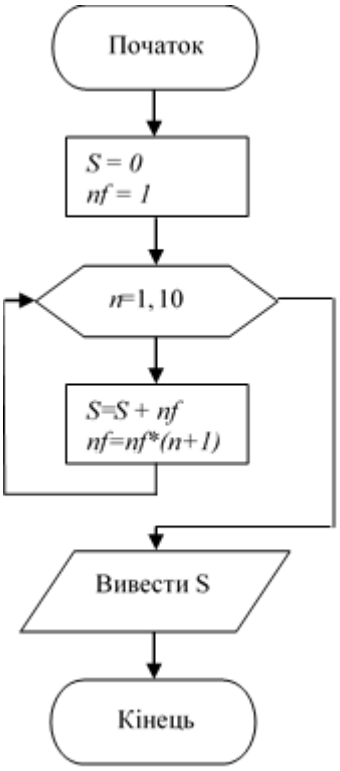


Рис. 4.9 – Алгоритм розрахунку суми факторіалів

Як бачимо, на початку алгоритму в блоці підготовки до циклу змінні отримують свої початкові значення. Оскільки кількість елементів для складення чітко визначена, використано цикл з відомою кількістю повторювань. В тілі циклу спочатку відбувається чергове накопичення суми, а потім розраховується нове значення факторіалу, яке буде використане на наступній ітерації циклу.

Однією з поширених задач, в якій використовуються рекурентні формули, є розрахунок значень функцій за допомогою їх розкладення у ступеневі ряди. Відомо, що будь-яку функцію можна наближено представити у вигляді деякого ряду. Підставивши значення аргументу, можна знайти часткову суму ряду і вважати її значенням функції від того самого аргументу з певною точністю наближення. Чим більша точність потрібна, тим більший відрізок ряду треба буде обчислювати.

Приклад 4.8. Обчислення суми нескінченого ряду.

Однією з поширених задач, в якій використовуються рекурентні формули, є розрахунок значень функцій за допомогою їх розкладення у ступеневі ряди. Відомо, що будь-яку функцію можна наближено представити у вигляді деякого ряду. Підставивши значення аргументу, можна знайти часткову суму ряду і вважати її значенням функції від того самого аргументу з певною точністю наближення. Чим більша точність потрібна, тим більший відрізок ряду треба буде обчислювати.

Зверніть увагу!

Для обчислень в програмі можна використовувати будь-який з циклів. Головне правило – це враховувати в сумі тільки ті елементи, які необхідні для вирішення задачі!

Необхідно обчислити суму нескінченого ряду з точністю до члена ряду, меншого ε (0<ε<1).

$$S = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!} + \ldots = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$

Рішення:

Позначимо поточний член ряду як $U_n = (-1) \cdot \frac{x^{2n}}{(2n)!}$

Для обчислення поточного члена ряду використаємо рекурентну формулу:

$$U_n = \lambda * U_{n-1}.$$

Коефіцієнт λ обчислюється таким чином:

$$\lambda = \frac{U_n}{U_{n-1}} = \frac{(-1)^n \cdot x^{2n}}{(2n)!} \cdot \frac{[2\cdot(n-1)]!}{(-1)^{n-1} \cdot x^{2(n-1)}} =$$
$$= \frac{(-1)^n \cdot x^{2n}}{(2n)!} \cdot \frac{(2n-2)!}{(-1)^{-1} \cdot (-1)^n \cdot x^{2n} \cdot x^{-2}} =$$
$$= -\frac{x^2 \cdot (2n-2)!}{(2n-2)!\cdot(2n-1)\cdot 2n} = -\frac{x^2}{2n\cdot(2n-1)}.$$

Початкове значення елемента ряду:

$$U_0 = (-1)^0 \frac{x^0}{0!} = 1.$$

Початкове значення суми:

$$S_{поч} = 0$$

Фрагмент програми для розрахунку суми із заданою точністю ε:

```
float u;
float x;
int n;
float S;
float eps;

do{
    cout<<”x=”;
    cin>>x;
    if (x==0)
        cout<<”Incorrect value x”<<endl;
}while (x==0);

do{
    cout<<”eps=”;
    cin>>eps;
    if (eps<=0 || eps>=1)
        cout<<”Incorrect value eps”<<endl;
}while (eps<=0 || eps>=1);

S = 0.0;
n = 0;
u = 1;
do{
    S+=u;
    n++;
    u*=-x*x/(2*n)/(2*n-1);

}while (abs(u)>eps);

cout<<”S=”<<S<<endl;
cout<<”n=”<<n<<endl;
```

Зверніть увагу!

Для обчислень в програмі використовується цикл do…while. Це призводить до того, що в сумі буде врахований мінімум один член ряду. Якщо проаналізувати саму послідовність та вимоги до точності, то можна побачити, що перший елемент

завжди дорівнює 1 (він не залежить від значення x) і повинен бути врахованим в сумі. Для задач з іншими формулами для обчислень цикл do…while може не підійти.

Контрольні питання

1. Що розуміють під лінійними операторами в програмуванні?
2. Чим лінійна структура програми відрізняється від розгалуженої та циклічної?
3. Які оператори в C++ належать до лінійних?
4. Що таке оператор присвоєння і яке його призначення?
5. Наведіть приклади задач, які реалізуються за допомогою лінійних алгоритмів.
6. Яке призначення умовного оператора if?
7. Яка структура оператора if–else?
8. Коли доцільно використовувати вкладені умови?
9. Що таке умовний оператор ‘?:’?
10. Які обмеження має тернарний оператор?
11. Яке призначення оператора switch?
12. Чим switch відрізняється від послідовності if–else?
13. Які помилки виникають при використанні умовних операторів?
14. Наведіть приклади задач, які реалізуються за допомогою алгоритмів розгалуження.
15. Яке призначення циклів у програмі?
16. Які типи циклів підтримує C++?
17. У чому різниця між циклами for, while і do while?
18. Коли доцільно використовувати кожен тип циклу?
19. Яке призначення операторів break і continue?
20. Що таке вкладені цикли?
21. Які помилки призводять до нескінченних циклів?
22. Як перевірити коректність роботи циклу?

5. РОБОТА ЗІ СКЛАДНИМИ СТРУКТУРАМИ ДАНИХ

5.1 Перелік

Тип перелік у мові C++ використовується для задання скінченного набору іменованих значень, що логічно пов'язані між собою. Переліки підвищують читабельність коду, зменшують кількість помилок і роблять програму більш самодокументованою. Перелік визначає новий користувацький тип даних, значеннями якого можуть бути лише явно перелічені константи. Приклади застосування переліків:

- стани системи;
- дні тижня;
- рівні доступу;
- коди результатів.

Синтаксис переліку enum в C++ наступний:

enum *назва_типу {елементи_переліку} декларатор;*

де

- *назва_типу* – ідентифікатор нового користувацького типу ;
- *елементи_переліку* – іменовані цілочисельні константи;
- *декларатор* – ідентифікатор змінної, яка має об'явлений користувацький тип, необов'язковий параметр.

Наприклад, список з днів тижня може бути представлений наступним переліком:

```
enum Day {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
};
```

Зверніть увагу!

Значення константам можна присвоїти тільки при об'явленні, змінити їх в коді неможливо.

Імена всіх констант повинні бути унікальними.

Значення констант можуть співпадати.

Константи можуть бути неявно перетворені в int, але int ніколи не неявно перетворюється на значення enum.

За замовчуванням першому елементу присвоюється 0, кожному наступному – значення, збільшене на 1. В цьому прикладі Monday=0, Tuesday=1 і т.д.

Значення елементам переліку можна задати явно, наприклад:

```
enum ErrorCode {
    Ok = 0,
    FileNotFound = 404,
    AccessDenied = 403
};
```

Якщо відбувається явне визначення значень, то наступні елементи без явного значення продовжують відлік від попереднього:

```
enum notes{ DO = 1, RE, MI, FA, SOL, LA, SI };
enum Suit { Diamonds = 5, Hearts, Clubs = 4, Spades = Hearts + 3};
```

В цьому прикладі в переліку notes RE=2, MI=3 і т.д. В переліку Suit маємо Hearts=6 (оскільки попередня константа Diamonds = 5), таким чином, Spades=6+3=9.

Зверніть увагу!

У звичайному enum усі перелічувачі потрапляють у загальну область видимості, що може спричиняти конфлікти імен:

enum Color { Red, Green, Blue };

*enum Signal { **Red**, Yellow, Green }; // помилка, співпадає з вже оголошеною константою в переліку Color*

Для усунення проблем з видимістю імен починаючи з C++11, рекомендовано використовувати переліки з областю видимості:

enum class *назва_типу {елементи_переліку};*

Особливості enum class:

- елементи переліку мають власну область видимості;
- доступ здійснюється через ім’я типу (*назва_типу::елемент_переліку*);
- немає неявного перетворення в int;
- підвищена типобезпечність.

Приклад переліку з областю видимості:

```
enum class Color {
    Red,
    Green,
    Blue
};
```

5.2 Масиви фіксованої довжини

Масив у мові C++ - це структура даних, що зберігає набір елементів одного типу, розміщених у пам’яті послідовно, до яких можна звертатися за допомогою індексу.

Кожен елемент масиву має:

- однаковий тип даних;

– власний порядковий номер (індекс);
– фіксоване місце в пам’яті.

У масиви можна об’єднувати результати експериментів, списки прізвищ співробітників, різні складні структури даних. Наприклад, список студентів у журналі є масивом. У масиві дані розрізняються за своїми порядковими номерами (індексами). Якщо кожний елемент масиву визначається за допомогою одного номера, то такий масив називається одновимірним, якщо за двома – то двовимірним. Двовимірний масив – це таблиця з рядків і стовпців. У таблицях перший номер вказує на рядок, а другий – на положення елемента в рядку. Усі рядки таблиці мають однакову довжину. Одновимірний масив може бути набором чисел, сукупністю символьних даних чи елементів іншої природи (навіть масив масивів). Фізична структура масиву – це спосіб розміщення елементів масиву в пам’яті комп’ютера. Під елемент масиву виділяється кількість байт пам’яті, яка визначається базовим типом елемента цього масиву. Кількість елементів масиву і розмір базового типу визначають розмір пам’яті для зберігання масиву. Елементи масиву розташовуються у пам’яті комп’ютера підряд, один за одним.

Сама найважливіша операція фізичного рівня над масивом – доступ до заданого елемента. Як тільки реалізовано доступ до елемента, над ним може бути виконана будь-яка операція, що має сенс для того типу даних, якому відповідає елемент. Перетворення логічної структури масиву у фізичну називається процесом лінеаризації, в ході якого багатовимірна логічна структура масиву перетвориться в одновимірну фізичну структуру.

Адресою масиву є адреса першого байту початкового компоненту масиву. Індексація елементів масиву у C++ починається з нуля. Для масиву з N елементів допустимі індекси від 0 до N-1. Звернення за межі масиву призводить до невизначеної поведінки.

5.2.1 Одновимірні масиви

Синтаксис оголошення одновимірного масиву фіксованої довжини:

тип ім’я_масиву[розмір];

При такому способі оголошення розмір масиву фіксується під час компіляції і не може бути змінений під час виконання програми.

Приклад оголошення масиву з 5 цілочисельних елементів:

```
int numbers[5];
```

В цьому прикладі:

– int - тип елементів масиву;
– numbers - ім’я масиву;

– - кількість елементів (довжина масиву).

Як і прості змінні, масиви можуть бути ініційовані при оголошенні. Ініціалізатор для об'єктів складових типів (яким є масив) складається зі списку ініціалізаторів, розділених комами і укладених у фігурні дужки. Кожен ініціалізатор в списку являє собою або константу відповідного типу, або, у свою чергу, список ініціалізаторів. Ця конструкція використовується також для ініціалізації багатовимірних масивів.

Наявність списку ініціалізаторів в оголошенні масиву дозволяє не вказувати число елементів по його першій розмірності. У цьому випадку кількість елементів у списку ініціалізаторів і визначає число елементів по першій розмірності масиву. Тим самим визначається розмір пам'яті, яка необхідна для зберігання масиву. Число елементів по решті розмірностей масиву, окрім першої, вказувати обов'язково.

Якщо в списку ініціалізаторів менше елементів, ніж у масиві, то залишилися елементи неявно ініціюються нульовими значеннями. Якщо ж число ініціалізаторів більше, ніж потрібно, то видається повідомлення про помилку.

Приклади різних оголошень та ініціалізації одновимірного масиву:

```
//Просте оголошення:
int x [10]; // Одновимірний масив з 10 цілих чисел int N=10;
int y [N]; //Помилка – розмір масиву не може бути визначений через змінну

//Оголошення з ініціалізацією:
int a[3] = {0, 1, 2}; // кількість ініціалізаторів дорівнює кількості елементів
double b[5] = {0.1, 0.2, 0.3}; // кількість ініціалізаторів менше кількості елемен
тів, інші елменти ініціюються нулями
int c [] = {1, 2, 4, 8, 16};// кількість елементів масиву визначається за
кількістю ініціалізаторів
int e[3] = {0, 1, 2, 3}; // Помилка - кількість ініціалізаторів більше кількості
елементів
```

Не існує операції присвоювання масиву, відповідного описаному раніше способу ініціалізації:

```
int a[3] = {0, 1, 2}; // Оголошення і ініціалізація
int b[3];
b = {0, 1, 2}; // Помилка
```

Оскільки масив є індексованою структурою, то звернення до елементу відбувається із використанням його номеру. Синтаксис звернення до елементу одновимірного масиву:

ім'я_масиву [цілочисельний_вираз]

Тут квадратні дужки є вимогою синтаксису мови, а не ознакою необов'язковості конструкції. Індекс масиву може бути не тільки константою, а й виразом, який має цілочисельний тип, наприклад, a[i + 1] (тут ***a*** повинно бути ім'ям раніше оголошеного масиву, а ***i*** - змінної цілого типу).

Для обробки елементів масиву зазвичай використовується оператор покрокового циклу for:

for (i = 0; // Присвоюємо лічильнику циклу значення індексу першого елемента

i<n; // Умова продовження циклу - поки значення лічильника менше кількості елементів масиву

i++) // Збільшуємо лічильник циклу на 1 для переходу до наступного елементу масиву

<тіло циклу> // У тілі циклу відбувається обробка одного елемента масиву

Припустимо, ми хочемо отримати масив ***x***, як на рис.5.1.

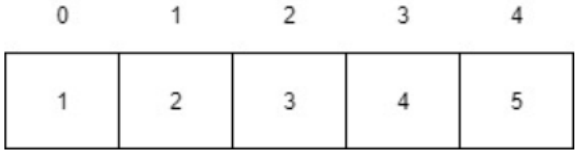


Рис.5.1 – Бажана структура масиву ***x***

Для створення масиву та заповнення значеннями 1, 2, 3, 4, 5 будемо використовувати нижченаведений код:

```
int x [5];
for (int i=0; i<5; i++)
     x[i]=i+1;
```

В цьому коді:

- x – ім’я масиву;
- i – номер елементу в масиві x;
- x[i] – значення, яке зберігається в масиві x під номером i.

Слід зауважити, що у мові C++ немає можливості вводити і виводити весь масив одним оператором вводу/виводу (крім рядків). Можна вводити і виводити тільки один елемент масиву. Отже, для того щоб ввести весь масив, треба використовувати цикл.

Однією з особливостей вирішення задач на обробку одновимірних масивів є те, що ми ***можемо розділити введення даних та їх обробку***. Крім того, в масиві, на відміну від послідовності, представленої однією коміркою пам’яті, ми ***можемо змінювати та обробляти дані, застосовуючи нелінійні схеми обробки***.

Лінійна схема обробки масиву практично така сама, як і у випадку обробки послідовності.

Нелінійні схеми залежать від методу вирішення задачі і в кожному окремому випадку різні.

Важливо!

Якщо схема обробки послідовності елементів є лінійною, то замість масиву з метою економії пам'яті можна використовувати одну буферну змінну для збереження тільки поточного значення послідовності

Лінійна схема обробки одновимірного масиву виглядає наступним чином (рис.5.2)

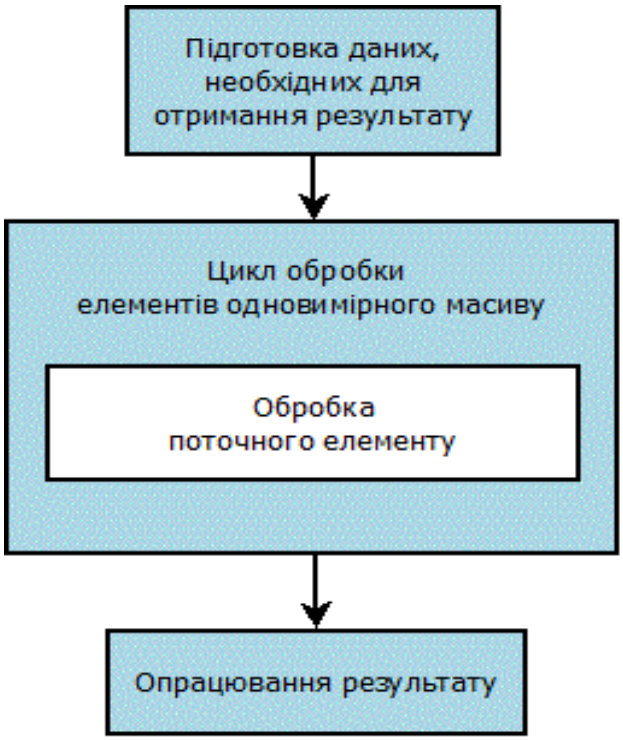


Рис.5.2 – Лінійна схема обробки елементів масиву

В залежності від задачі, блоки підготовки даних, необхідних для отримання результату та блоки опрацювання результату можуть бути відсутніми, але цикл обробки поточного елементу, як правило, є обов'язковим. Сама конструкція циклу може бути різною та залежить від наступних факторів:

- скільки елементів масиву обробляється;
- з яким кроком необхідно пересуватись по масиву;
- в якому напрямку йде пересування по масиву;
- що є умовою продовження/припинення обробки масиву.

Дані наведено окремі приклади вирішення типових задач, які можуть бути використані, як базові блоки в інших, більш складних завданнях.

Приклад 5.1. Введення елементів одновимірного масиву з N елементів.

```
//блок підготовки даних для отримання результату відсутній
for (int i = 0; i < N; i++) //цикл обробки елементів одновимірного масиву
{
	cout << "a" << i + 1 << " = "; //обробка поточного елементу масиву
	cin >> a[i];                   // полягає в його введенні

}
//блок опрацювання результату відсутній
```

Приклад 5.2. Заповнити одновимірний масив довжиною N числами починаючи з числа K з кроком h.

```
	// блок підготовки полягає у попередньому формуванні значень чисел K та h,
	// вони можуть буди генеровані, введені або розраховані

	for (int i = 0; i < N; i++) //цикл обробки елементів одновимірного масиву
		a[i] = K + i*h; //обробка поточного елементу масиву – формування значення за формулою

	//блок опрацювання результату може полягати у виведенні масиву чи в подальшому його використанні для інших задач
```

Приклад 5.3. Виведення в стовпчик елементів одновимірного масиву довжиною N

```
	//блок підготовки полягає у попередньому формуванні елементів масиву будь яким способом – введення, генерація, розрахунок, тощо
	for (int i = 0; i < N; i++) //цикл обробки елементів одновимірного масиву
		cout << "a[" << i + 1 << "] = " << a[i] << endl; //обробка поточного елементу масиву – виведення підказки та самого значення елементу
	//блок опрацювання результату відсутній
```

Приклад 5.4. Сума елементів одновимірного масиву з N елементів

```
	S = 0;// підготовка початкового значення для розрахунку суми

	for (int i = 0; i < N; i++) //цикл для обробки елементів одновимірного масиву
		S += a[i];		//обробка поточного елементу – додавання його до суми


	cout << "Sum = " << S <<endl;//опрацювання результату – виведення значення суми
```

Приклад 5.5. Кількість позитивних елементів одновимірного масиву з N елементів

```
	k = 0;// підготовка початкового значення для розрахунку кількості

	for (int i = 0; i < N; i++) //цикл для обробки елементів одновимірного масиву
		if (a[i] > 0)		//обробка поточного елементу – перевірка елементу та
			k++;		//зміна значення кількості k

	cout << "Positive numbers = " << k <<endl;//опрацювання результату – виведення кількості
```

Приклад 5.6. Мінімум одновимірного масиву з N елементів

```
	min = a[0];// підготовка початкового значення для розрахунку мінімуму

	for (int i = 1; i < N; i++) //цикл для обробки елементів одновимірного масиву
	// елемент з номером 0 не обробляється повторно, починаємо з наступного
	if (min < a[i])		//обробка поточного елементу – перевірка елементу та
			min = a[i];	//зміна значення мінімуму

	cout << "Min =" << min <<endl; //опрацювання результату – виведення мінімуму
```

Приклад 5.7. Замінити в одновимірному масиві з N елементів нулі на одиниці

```
	//блок підготовки результату полягає у формуванні елементів масиву: введення, генерація тощо

	for (int i = 0; i < N; i++) //цикл для обробки елементів одновимірного масиву
```

```
        if (a[i] == 0)      //обробка поточного елементу – перевірка елементу та
                    a[i] = 1;     //зміна його значення

        //блок опрацювання результату може полягати у виведенні масиву чи в подальшому його використанні для інших задач
```

5.2.2 Багатовимірні масиви. Матриці

Робота з багатовимірними масивами фіксованої довжини в мові С++ є природним розширенням роботи з одновимірними масивами та використовується для подання табличних, матричних і багатовимірних структур даних, розмір яких відомий на етапі компіляції. Такі масиви широко застосовуються в алгоритмах обробки зображень, чисельних методах, моделюванні та системному програмуванні.

Найпоширенішим випадком є двовимірний масив, який логічно можна уявити як таблицю з рядків і стовпців. Для двовимірних масивів часто використовують назву «матриця».

У С++ двовимірний масив оголошується як масив масивів, тобто кожен його елемент першого рівня є окремим одновимірним масивом. Наприклад, масив int a[3][4] містить три рядки по чотири елементи в кожному. Перший індекс визначає номер рядка, другий — номер стовпця, при цьому індексація, як і завжди в С++, починається з нуля. Таким чином, звернення a[1][2] означає доступ до третього елемента другого рядка.

Синтаксис оголошення двовимірного масиву фіксованої довжини:

назва_типу ідентифікатор [цілочисельна_константа1] [цілочисельна_константа2];

Тут:

назва_типу – тип елементу масиву;

ідентифікатор – ім’я масиву;

цілочисельна_константа1 – кількість рядків масиву;

цілочисельна_константа2 – кількість стовпчиків масиву.

Приклади оголошень двовимірного масиву:

```
//Просте оголошення:
int x [10][5]; // Двовимірний масив з цілих чисел розміром 10x5

//Оголошення з ініціалізацією:
int matrix1[3][5] = {
    { 1, 2, 3, 4, 5 },
    { 2, 4, 6, 8, 10 },
    { 3, 6, 9, 12, 15 }
};

//ініціалізація прямокутного масиву
int matrix2[][4] = {
    { 1, 2, 3, 4 },
    { 5, 6, 7, 8}
};// при наявності ініціалізатору, самий лівий розмір масиву може бути опущений
```

Синтаксис звернення до елементу двовимірного масиву:

ім'я_масиву [цілочисельний_вираз1][цілочисельний_вираз2]

Багатовимірний масив обробляється за допомогою вкладених циклів. Як правило (але не обов’язково!), кількість циклів збігається з розмірністю масиву, тобто для обробки матриці, наприклад, нам, як правило, знадобиться 2 вкладених цикли. Зверніть увагу, що в деяких задачах для обробки матриці може знадобитися лише один цикл або більше, ніж два цикли – для визначення кількості циклів треба аналізувати порядок обробки даних в матриці.

Базова схема обробки двовимірного масиву представлена на рис.5.3. для застосування схеми введемо рівні результатів:

– Результат рівня 0 – формується один для всього двовимірного масиву;
– Результат рівня 1 – формується один для поточного рядка/стовпчика двовимірного масиву;
– Результат рівня 2 – формується один для поточного елементу двовимірного масиву.

Рис.5.3 – Базова схема обробки двовимірного масиву

Алгоритм застосування схеми на рис. 5.3:

1. Аналізуємо, скільки результатів повинно утворитися при вирішенні задачі та визначаємо їх рівень.
2. Аналізуємо, що треба для ініціалізації результату відповідного рівня.
3. Визначаємо необхідний порядок циклів: якщо обробка ведеться по рядках (традиційний порядок обробки), то зовнішній цикл змінює індекси рядків, а внутрішній – індекси стовпчиків, і навпаки.

Обробка по рядках:

```
for (int i =.....) {
      for (int j = .....) {
        .......a[i][j]...
      }
  }
```

Обробка по стовпчиках:

```
for (int j = .....){
      for (int i = .....) {
        ....... a[i][j]...
      }
  }
```

4. Визначаємо, які операції необхідно виконати з отриманим результатом (результатами).

Приклад 5.8. Розрахувати суму елементів матриці A розміром MxN.

Рішення:

1. Аналізуємо, який результат ми маємо отримати: він ***один для всієї матриці***, тому застосовуємо схему для формування результату рівня 0.

2. Записуємо оператори, що формують суму (використовуємо метод накопичення):

S=0;// підготовка результату

…

S+=A[i][j];//обробка поточного елементу та формування результату

…

cout <<"S=“<<S;//обробка результату в даному прикладі полягає у виведенні його на екран

3. Визначаємо порядок обробки матриці – в даній задачі байдуже, яким способом перебирати елементи, тому використовуємо класичну схему обробки по рядках. Обробляється вся матриця, тому діапазони індексів змінюються від 0 до максимального значення розмірності.

4. Обробка результату в даному прикладі полягає у виведенні його на екран. Оскільки результат утворюється один для всієї матриці, то виводимо його поза циклами обробки матриці.

```
S=0;// підготовка результату
for (int i = 0; i < M; i++) { // цикл 1
        for (int j = 0; j < N; j++) { // цикл 2
            S+=A[i][j];//обробка поточного елементу та формування результату
        }
    }
cout <<"S="<<S;//обробка результату в даному прикладі полягає у виведенні його на
екран
```

Приклад 5.9. Розрахувати суми по рядках прямокутної матриці розміром MxN.

1. Аналізуємо, який результат ми маємо отримати: нам потрібен ***окремий результат для кожного рядка***, тому застосовуємо схему для формування результату рівня 1.

2. Записуємо оператори, що формують результат (використовуємо метод накопичення):

S=0;// підготовка результату

…

S+=A[i][j];//обробка поточного елементу та формування результату

…

3. Визначаємо порядок обробки матриці – в даній задачі нам необхідно отримати результати для рядків, тому зовнішній цикл повинен змінювати індекс рядка, а внутрішній – індекс стовпчика. Обробляється вся матриця, тому діапазони індексів змінюються від 0 до максимального значення розмірності.

Зверніть увагу, що оператори підготовки результату та обробки результату повинні розташовуватись в межах циклу 1, але за межами циклу 2!

4. Операції, які необхідно виконати з результатами – виведення на екран.

```
for (int i = 0; i < M; i++) { // цикл 1
        S=0;// підготовка результату
        for (int j = 0; j < N; j++) { // цикл 2
            S+=A[i][j];//обробка поточного елементу та формування результату
        }
        cout <<"S"<<i<<"="<<S<<endl;//обробка результату полягає у виведенні його
на екран
    }
```

Приклад 5.10. Розрахувати суми по стовпчиках прямокутної матриці розміром MxN.

Рішення аналогічно прикладу 5.9, відрізняється тільки порядок обробки матриці, а відповідно – порядок слідування циклів):

1. Аналізуємо, який результат ми маємо отримати: нам потрібен ***окремий результат для кожного стовпчика***, тому застосовуємо схему для формування результату рівня 1.

2. Записуємо оператори, що формують результат (використовуємо метод накопичення)

Зверніть увагу, що оператори підготовки результату та обробки поточного елементу для формування суми не змінились у порівнянні з прикладом 5.9!

S=0;// підготовка результату

…

S+=A[i][j];//обробка поточного елементу та формування результату

…

3. Визначаємо порядок обробки матриці – в даній задачі, на відміну від прикладу 5.9, нам необхідно отримати результати для стовпчиків, тому зовнішній цикл повинен змінювати індекс стовпчика, а внутрішній – індекс рядка. Обробляється вся матриця, тому діапазони індексів змінюються від 0 до максимального значення розмірності. Зверніть увагу, що оператори підготовки результату та обробки результату розташовуються в межах циклу 1, але за межами циклу 2.

4. Операції, які необхідно виконати з результатами – виведення на екран.

```
for (int j = 0; j < N; j++) { // цикл 1
        S=0;// підготовка результату
        for (int i = 0; i < M; i++) { // цикл 2
            S+=A[i][j];//обробка поточного елементу та формування результату
        }
        cout <<"S"<<j<<"="<<S<<endl;//обробка результату полягає у виведенні його
на екран
    }
```

Приклад 5.11. Розрахувати кількість від'ємних елементів в кожному стовпчику матриці A розміром MxN.

1. Аналізуємо, який результат ми маємо отримати: нам потрібен ***окремий результат для кожного стовпчика***, тому застосовуємо схему для формування результату рівня 1.

2. Записуємо оператори, що формують результат (використовуємо метод накопичення):

k=0;// підготовка результату

…

if(A[i][j]<0) k++;//обробка поточного елементу та формування результату

…

3. Визначаємо порядок обробки матриці – в даній задачі нам необхідно отримати результати для стовпчиків, тому зовнішній цикл повинен змінювати індекс стовпчика, а внутрішній – індекс рядка. Обробляється вся матриця, тому діапазони індексів змінюються від 0 до максимального значення розмірності. Зверніть увагу, що оператори підготовки результату та обробки результату розташовуються в межах циклу 1, але за межами циклу 2.

4. Операції, які необхідно виконати з результатами – виведення на екран.

```
for (int j = 0; j < N; j++) { // цикл 1
      k=0;// підготовка результату
      for (int i = 0; i < M; i++) { // цикл 2
         if(A[i][j]<0)   k++;//обробка поточного елементу та формування результату
        }
      cout <<"кількість від'ємних елементів в стовпчику "<<j<<"складає"<<k<<endl;
//обробка результату полягає у виведенні його на екран
    }
```

Приклад 5.12. замінити в матриці A розміром MxN всі елементи більші за число X на це число.

1. Аналізуємо, який результат ми маємо отримати: нам потрібно опрацювати кожне число в матриці, але без формування підсумків по стовпчикам/рядкам або по всій матриці, тому застосовуємо схему для формування результату рівня 2.

2. Записуємо оператори, що формують результат:

if(A[i][j]>X) A[i][j]=X; //обробка поточного елементу та формування результату

3. Визначаємо порядок обробки матриці – в даній задачі байдуже, яким способом перебирати елементи, тому використовуємо класичну схему обробки по рядках. Обробляється вся матриця, тому діапазони індексів змінюються від 0 до максимального значення розмірності.

4. Після виконання замін в матриці, її можна вивести на екран або використовувати для подальших розрахунків, цей блок відсутній у наведеному коді.

```
for (int i = 0; i < M; i++) { // цикл 1
        for (int j = 0; j < N; j++) { // цикл 2
             if(A[i][j]>X) A[i][j]=X; //обробка поточного елементу та формування р
езультату
        }
}
```

Слід зауважити, що попри універсальність схеми для більшості типових задач, існує чимало випадків, для яких вона не дозволяє напряму отримати рішення, а передбачає гнучке застосування цієї схеми, а в окремих випадках, навіть схеми обробки одновимірних масивів:

- може знадобитися більше циклів;
- може знадобитися менше циклів;
- матриця обробляється не вся;
- матриця обробляється вся, але за складною схемою;
- матриця взагалі представляється не двовимірним масивом.

Нам знадобиться більше циклів якщо:

1) Необхідно пройтися декілька разів по одному й тому ж рядку/стовпчику, перед тим, як перейти до наступного. Наприклад:
   - замінити нульові елементи кожного рядка матриці на середнє арифметичне рядка;
   - відсортувати елементи кожного стовпчика за зростанням.
2) Напрям обробки елементів матриці змінюється в різних вимірах. Наприклад:
   - заповнити матрицю по спіралі числами починаючи з 1, початкова точка – правий нижній кут, напрям руху – проти годинникової стрілки.

Так само, нам знадобиться менше циклів, якщо в рядку/стовпчику обробляються лише окремі елементи з відомими індексами, а не весь рядок/стовпчик. Наприклад:

- в квадратній матриці a порядку n замінити нулями елементи на головній діагоналі;
- обчислити в кожному рядку матриці суму першого та останнього елементу;
- поміняти в матриці місцями перший та останній елементи в кожному стовпчику.

Якщо матриця обробляється не вся, то необхідно визначити, який саме фрагмент обробляється, та які зі схем можна до цих фрагментів застосувати.

Так само, якщо матриця обробляється вся, але за складною схемою, то спочатку виділяємо фрагменти, які обробляються за різними схемами, а далі для кожного фрагменту реалізуємо свою схему.

В деяких задачах матричні дані реального світу можуть бути представлені взагалі не двовимірним масивом. Наприклад, розріджений масив, більшість елементів якого є нулями може бути представлений трійками (два індекси та значення на перетині), в зображенні блоки послідовних пікселів однакового кольору можуть бути закодовані номером кольору та кількістю таких пікселів в послідовності і т.д.

Нижче наведені приклади застосування схеми на рис.5.3 до деяких вищезазначених «проблемних» задач.

Приклад 5.13. Замінити нульові елементи кожного рядка матриці на середнє арифметичне рядка.

1. Для вирішення задачі треба ***обробити кожен рядок 2 рази*** та виконати наступне:

1.1 Знайти середнє арифметичне поточного i-го рядка. Фрагмент коду для розрахунку:

```
Avg=0;
    for (int j = 0; j < N; j++)
        Avg+=A[i][j];
    Avg/=N;
```

1.2 Замінити нульові елементи поточного i-го рядка на знайдене середнє арифметичне цього рядка. Фрагмент коду для заміни:

```
  for (int j = 0; j < N; j++)
      if (A[i][j]==0) A[i][j]=Avg;
```

2. Визначаємо порядок обробки матриці – в даній задачі нам необхідно отримати результати для рядків, тому зовнішній цикл повинен змінювати індекс рядка, а внутрішні цикли – індекс стовпчика. Обробляється вся матриця, тому діапазони індексів змінюються від 0 до максимального значення розмірності.

3. Після виконання замін в матриці, її можна вивести на екран або використовувати для подальших розрахунків.

```
for (int i = 0; i < M; i++) { //цикл 1
//перший етап обробки рядка – обчислення середнього арифметичного
    Avg=0;
    for (int j = 0; j < N; j++)
         Avg+=A[i][j];
    Avg/=N;
//другий етап обробки рядка – заміна нулів на значення Avg
    for (int j = 0; j < N; j++)
        if (A[i][j]==0) A[i][j]=Avg;
}
//блок обробки результатів визначається подальшими вимогами в програмі
```

Приклад 5.14. В квадратній матриці A порядку N замінити нулями елементи на головній діагоналі.

1. Аналізуємо, який результат ми маємо отримати: нам потрібно опрацювати ***кожне число на головній діагоналі матриці***, але без формування підсумків по стовпчикам/рядкам або по всій матриці, тому можемо застосувати схему для формування результату рівня 2.

Якщо проаналізувати індекси цих елементів (A[0][0], A[1][1], A[2][2] …), то бачимо закономірність: номер стовпчика та номер рядка співпадають. Таким чином, ми можемо використовувати ***один індекс для позначення як номера рядка, так і номера стовпчика***.

2. Записуємо оператор, що формує результат:

A[i][i]=0; //обробка поточного елементу та формування результату

3. Визначаємо порядок обробки матриці – в даній задачі будемо змінювати номер рядка для руху по головній діагоналі матриці. Обробляються всі рядки матриці, тому діапазон індексу змінюється від 0 до максимального значення розмірності. ***Зверніть увагу, що для обробки використовується один цикл!***

4. Після виконання замін в матриці, її можна вивести на екран або використовувати для подальших розрахунків.

```
for (int i= 0; i < N; i++)  // цикл для обробки елементів на головній діагоналі
         A[i][i]=0; //обробка поточного елементу та формування результату
//блок обробки результатів визначається подальшими вимогами в програмі
```

Приклад 5.15. Обчислити в кожному рядку матриці суму першого та останнього елементу.

1. Аналізуємо, який результат ми маємо отримати: нам потрібно опрацювати в ***кожному рядку по два елементи, при цьому номери цих елементів залежать лише від номеру рядка!***

Якщо проаналізувати індекси елементів, які треба сумувати, то маємо наступне:

A[0][0]+A[0][N]

A[1][0]+A[1][N]

A[2][0]+A[2][N]

…

Для довільного і-го рядка маємо закономірність A[i][0]+A[i][N]. Таким чином, ми можемо використовувати ***лише один індекс для розрахунку результатів, і, відповідно, один цикл! (по суті застосовується схема обробки лінійного масиву чи послідовності)***

2. Обробляються всі рядки матриці, тому діапазон індексу змінюється від 0 до максимального значення розмірності.

```
for (int i= 0; i < M; i++)  // цикл для обробки рядків
        cout<< 'S'<<i<<'='<<A[i][0]+A[i][N]<< endl; //обробка поточного елементу
та формування результату (виведення на екран)
```

5.4 Структури struct

Для обробки даних, які мають складну гетерогенну структуру, в C++ використовуються структури struct. Структури використовуються для представлення об’єктів, які мають певний логічний зв’язок між елементами. Структура в загальному випадку – це сукупність різнотипних елементів, яким присвоюється одне ім’я (воно може бути відсутнім), що займає одну ділянку пам’яті. Елементи, що складають структуру, називаються полями. Структура по суті визначає новий тип даних, який складається з полів (членів структури). Кожне поле має власний тип і ім’я, а вся структура розглядається як єдине ціле. Формат опису структури:

struct *ім’я_типу*

{

об’явлення змінних, що входять в структуру

} *список_змінних*;

При оголошенні структури можна вказувати *ім’я_типу*, за допомогою якого можна потім в програмі створювати структури. *список_змінних* визначає змінні, які створюються при об’явленні структури. Хоча б один із зазначених елементів обов’язково необхідно вказати при об’явленні структури.

Після того, як структура визначена, можна створювати змінні типу структури - об'єкти структури. Наприклад:

```
struct student

{ char pip [25];            // ПІП студента
  float avg;                    // середній бал
}  s1, s2;
```

Змінні s1 і s2 можна оголосити окремим оператором, наприклад:

```
struct student s1, s2;
```

Ініціювання полів структури слід здійснювати або при її описі, або в тілі програми. При описі структури ініціювання полів виглядає, наприклад, так:

```
s1 = { "Петренко П. П.", 3.5};
```

Для звернення до полів структури використовується точка, яку необхідно ставити після імені змінної:

ім'я_змінної.ім'я_поля

Якщо поле структури описано, як вказівник, то для звернення до його значення замість точки використовується ->.

Структури в C++ можна копіювати та присвоювати напряму:

```
student s2 = s1;
```

В цьому випадку виконується покомпонентне копіювання всіх полів.

Слід зазначити, що елементом структури може бути також структура, наприклад структура Person, що описує людину, містить поле дати народження birth, що представляється структурою Date:

```
struct Date {
    int day, month, year;
};

struct Person {
    string name;
    Date birth;
};
```

Доступ до полів відбувається в ієрархічному порядку відповідно до вкладеності структур:

```
Person p;
p.birth.year = 2000;
```

Структури можуть бути елементами масивів, наприклад, масив група, що складається з 30 структур типу Student:

```
Student group[30];
```

Доступ до полів здійснюється так само з урахуванням ієрархії вкладеності – спочатку необхідно звернутись до елементу масиву (таким чином, ми отримуємо доступ до структури), а потім до поля самої структури:

```
float mark= group[i].avg;
```

5.5. Робота з символьними та рядковими даними

У процесі розробки програм часто виникає потреба працювати з текстовою інформацією: іменами, повідомленнями, командами користувача, кодами, словами тощо. У мові C++ така інформація подається за допомогою символьних та рядкових даних.

Окремі символи у C++ зберігається у змінних типу char, константний символ записується в одинарних лапках:

```
char c = 'A';
```

Кожен символ має числовий код (зазвичай ASCII або його розширення). Це дозволяє порівнювати символи, виконувати арифметичні операції над ними, перевіряти діапазони (наприклад, чи є символ літерою).

При введенні символів варто треба враховувати, що звичайні методи та оператори потокового введення пропускають пробільні символи, тому, якщо є потреба врахувати при введенні всі символьні елементи, використовують спеціальні функції:

```
char c;
//cin >> c; якщо c буде містити пробільний символ, то він не зчитається цим операт
ором
cin.get(c);
```

Порівняння відбувається за числовими кодами символів. Наприклад, зважаючи на те, що символи цифр розміщені послідовно в таблиці кодів, можна перевіряти символ ***c*** на те, чи є він цифрою:

```
if (c >= '0' && c <= '9') {
    cout << "Цифра";
}
```

Для роботи з групами символів використовуються рядки. У процедурному програмуванні C++ класичним способом роботи з текстом є рядки символів, реалізовані як одновимірні масиви типу char. Таким чином, рядок визначеної довжини описується як:

char *ім'я_рядка[розмір+1];*

При оголошенні рядка фіксованої довжини завжди треба використовувати +1 до довжини, оскільки рядок – це послідовність символів типу char, яка завершується спеціальним нуль-символом '\0'. Цей символ визначає закінчення рядка. Всередині рядка може бути скільки завгодно таких символів, але компілятор виконує обробку до першого нуль-символу.

Таким чином, якщо ми маємо оголошення *char name[20];*, то це означає, що фактично ми звожемо зберігати в ньому тільки 19 символів.

Оголошення виду

```
char text[] = "Hello";
```

призведе до створення масиву, я в якому в пам'яті зберігається: 'H' 'e' 'l' 'l' 'o' '\0'.

Так само, як і при роботі за звичайними символами, при зчитуванні рядків відбувається зчитування до першого пробільного символу, тому використовують спеціальні методи, наприклад:

```
cin.getline(name, 20);
```

Оскільки робота з текстовими даними як правило включає доволі типові операції (пошук символу/фрагменту рядка, виділення фрагменту рядка, вирізання,

копіювання та інші), робота з рядковими даними передбачає використання спеціалізованої бібліотеки, що містить готові реалізації цих операцій. Оскільки реалізації функцій роботи з рядками постійно потерпають модифікацій з метою підвищення їх безпечності та якості роботи, в цьому посібнику наведено лише декілька таких функцій для прикладу. Повний перелік можна подивитись в довідковій документації відповідної версії мови C++.

Приклад 5.16. Є рядок з описом курсу. Необхідно:

- знайти слово "C++";
- якщо воно знайдене — замінити його на "C++ (procedural)";
- додати в кінець рядка припис " — 1 курс";
- перевірити, чи не збігається результат із заздалегідь заданим еталонним рядком.

```
#include <iostream>
#include <cstring>

using namespace std;

int main() {
    char text[100] = "Programming in C++";
    char insert[] = "C++ (procedural)";
    char suffix[] = " — 1 course";

    // 1. Пошук підрядка
    char* pos = strstr(text, "C++");

    if (pos != nullptr) {
        // 2. Видалення "C++"
        // зсуваємо кінець рядка вліво
        memmove(pos, pos + strlen("C++"), strlen(pos + strlen("C++")) + 1);

        // 3. Вставка нового підрядка
        memmove(pos + strlen(insert), pos, strlen(pos) + 1);
        strncpy(pos, insert, strlen(insert));
    }

    // 4. Додавання суфікса
    strcat(text, suffix);

    // 5. Порівняння з еталоном
    char expected[] = "Programming in C++ (procedural) — 1 course";

    if (strcmp(text, expected) == 0) {
        cout << "Рядок сформовано правильно\n";
    } else {
        cout << "Результат відрізняється\n";
    }

    cout << text << endl;
    return 0;
}
```

В цьому коді використано вказівники, тому, для розуміння програми необхідно прочитати відповідний розділ, який присвячений роботі з динамічною пам’яттю та вказівниками. Для більш «тонкої» обробки та повного контролю над пам’яттю, рядки можна обробляти в тому числі і як звичайні масиви.

Приклад 5.17. Дано рядок, що містить прізвище, ім'я та по батькові через один чи декілька пробілів. необхідно сформувати новий рядок, що містить прізвище та ініціали з крапками.

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

int main() {
    char fullName[] = "    Horbatiuk   Oksana     Leonidivna    ";
    char shortName[50];  // для прізвища та ініціалів
    int i = 0, j = 0;

    // Пропускаємо початкові пробіли
    while (fullName[i] == ' ') i++;

    // Копіюємо прізвище до пробілу
    while (fullName[i] != ' ' && fullName[i] != '\0') {
        shortName[j++] = fullName[i++];
    }

    // Додаємо пробіл між прізвищем та ініціалами
    shortName[j++] = ' ';

    // Пропускаємо пробіли перед ім'ям
    while (fullName[i] == ' ') i++;

    // Додаємо перший символ імені + крапка
    if (fullName[i] != '\0') {
        shortName[j++] = fullName[i++]; // перша літера імені
        shortName[j++] = '.';
    }

    // Пропускаємо пробіли перед по батькові
    while (fullName[i] == ' ') i++;

    // Пропускаємо символи імені
    while (fullName[i] != ' ' && fullName[i] != '\0') i++;

    while (fullName[i] == ' ') i++;

    // Додаємо перший символ по батькові + крапка
    if (fullName[i] != '\0') {
        shortName[j++] = fullName[i++]; // перша літера по батькові
        shortName[j++] = '.';
    }

    // Завершуємо рядок нуль-символом
    shortName[j] = '\0';

    cout << "Скорочене ім'я: " << shortName << endl;

    return 0;
}
```

Слід зауважити, що в сучасній розробці з рядками частіше за все працюють з використанням класу string та об’єктно-орієнтованого підходу.

5.6. Особливості використання вказівників в С++

5.6.1. Поняття вказівника, призначення та операції з вказівниками

Вказівник — це тип даних, який використовується для зберігання адрес змінних і об'єктів. Змінна типу вказівник зберігає значення адреса комірки пам'яті. Формат опису змінної-вказівника:

*тип * ім'я_змінної-вказівника;*

Приклади опису змінних-вказівників:

```
int *ptri; //вказівник на змінну цілого типу
char *ptrc; //вказівник на змінну символьного типу
float *ptrf; //вказівник на змінну дійсного типу.
```

Змінні різних типів займають різну кількість комірок пам'яті та по різному інтерпретуються, однак самі змінні типу вказівник мають однаковий розмір. Якщо оголошується декілька змінних-вказівників на один тип, то * ставиться перед кожним ім’ям. Наприклад, дана інструкція дозволяє оголосити три змінних-вказівника на комірки цілого типу:

```
int *ptr1, *ptr2, *prt3 ;
```

В наступному прикладі змінна ptr1 є вказівником, а змінна ptr2 – звичайною змінною цілого типу:

```
int *ptr1, ptr2;
```

Для отримання адреси комірки можна використовувати операцію &. Наприклад,

```
int v = 1;
int* ptr = &v; // ptr містить адресу змінної v
```

Для отримання значення, яке розташовано за деякою адресою, використовується операція непрямої адресації (розіменування вказівника) *.

Операція * дозволяє звертатися до змінної не напряму, а через вказівник, який містить адресу цієї змінної. Ця операція є одномісною і має асоціативність зліва на право. Операцію не слід плутати з бінарною операцією множення. Наприклад,

```
int v;
int* ptr = &v; // ptr містить адресу змінної v
*ptr = 1; // в комірку, на яку вказує ptr, записується значення 1
```

При роботі з вказівниками можна використовувати операцію присвоювання – при цьому присвоюються адреси:

```
int *ptr1, *ptr2;
int v;
ptr1 = &v;
*ptr1 = 1;
ptr2 = ptr1; //обидва вказівника вказують на одну й ту ж комірку пам’яті
```

При роботі з вказівниками можна використовувати наступні операції:

– p + n, де p - вказівник, n - ціле позитивне число, результат - деякий вказівник, отриманий зміщенням p на n позицій вправо;

– p-n, де p - вказівник, n - ціле позитивне число, результат - деякий вказівник, отриманий зміщенням p на n позицій вліво;

– p-q, де p і q – вказівники на один і той же тип, результат - ціле число, що дорівнює кількості кроків, на яке потрібно змістити q вправо, щоб він досяг вказівника p, також цей результат можна називати «відстанню» між вказівниками, воно може бути і негативним, якщо елемент, на який спрямований вказівник q розташований правіше (тобто, далі), ніж елемент, на який спрямований вказівник p;

– p ++ (інкремент), p-- (декремент), де p – вказівник, p = p + 1, p = p-1 відповідно.

Для виділення пам’яті під вказівники можна використовувати різні способи. Одним із них є застосування оператору new. Загальна форма оператору:

змінна-вказівник = new тип_змінної;

Наприклад:

```
int *a = new int; // Оголошення вказівника для змінної типу int
int *b = new float; // Оголошення вказівника для змінної типу float
```

Після застосування оператору new в динамічній пам’яті виділяється комірка відповідного типу. При використанні оператора new є можливість ініціалізації об'єкта:

змінна-вказівник = new тип_змінної (значення);

Наприклад:

```
int *a = new int(10); // Оголошення вказівника для змінної типу int
```

Для вивільнення пам’яті, яка була виділена оператором new, використовується оператор delete. Формат використання оператора:

delete змінна-вказівник;

Наприклад:

```
delete a;
```

5.6.2 Використання вказівників для роботи з динамічними масивами

При створенні будь-якого масиву в C ++ разом з ним створюється вказівник. Ім'я цього вказівника збігається з ім'ям масиву. Тип цього вказівника - «вказівник на базовий тип масиву». У цьому вказівнику зберігається адреса початкового елемента масиву. Щоб початок масиву не було втрачено, цей вказівник є константним, тобто його не можна направити на якийсь інший елемент масиву або записати туди адресу іншої змінної навіть відповідного типу, але можна скопіювати цю адресу в якийсь інший вказівник, який не є константним.

Для виділення пам’яті під масив за допомогою оператору new треба вказати розмір масиву:

змінна-вказівник = new тип_змінної [розмір_масиву];

Наприклад:

```
float *p = new float [10];
```

При виділенні пам’яті під масив одночасно ініціювати його не можна.

Приклад створення динамічного масиву та його заповнення числами від 1 до n:

```
#include <iostream>
using namespace std;
int main()
{
    int n; // розмір массиву
    cout << "Enter array size: ";
    cin >> n;

    int *a = new int[n]; // Виділення пам’яті під масив
    for (int i = 0; i < n; i++) {
        // Заповнення масиву і виведення значень його елементів
        a[i] = i+1;
        cout << "Value of " << i + 1 << " element is " << a[i] << endl;
    }
    delete [] a; // очищення пам'яті
    return 0;
}
```

З урахуванням того, що в масиві всі елементи розташовуються в пам'яті послідовно, почавши зі вказівника, спрямованого на початковий елемент, можна обійти всі елементи масиву, зміщуючи вказівник на кожному кроці вправо на мінімально можливу дистанцію (тобто, на сусідній праворуч елемент відповідного типу). Зсув вказівника може проводитися за допомогою операторів інкремента і декремента.

У наступному прикладі всі елементи масиву будуть виведені на екран без використання індексів (зверніть увагу, що при цьому параметри циклу можуть бути будь-якими, головне, щоб цикл виконався потрібну кількість разів):

```
int a[] = {-15, 13, 21, -12, 5, 132};
int* p = a;
```

```
for (int i = 11; i <= 16; i++) {
        cout << *p << endl;
        p++;
}
```

Для звернення до елементів масиву також можна використовувати тотожність

$$a[i] == *(a + i),$$

де a – вказівник на масив будь-якого типу, i – допустимий індекс цього масиву. Таким чином, фрагменти 1 та 2 еквівалентні:

```
//Фрагмент 1:
int a[] = {-15, 13, 21, -12, 5, 132};
for (int i = 0; i < 6; i++) {
        cout << * (a + i) << endl;
}
//Фрагмент 2:
int a[] = {-15, 13, 21, -12, 5, 132};
for (int i = 0; i < 6; i++) {
        cout << a[i] << endl;
}
```

При роботі з двовимірними динамічними масивами спочатку об’являється вказівник другого порядку, який посилається на масив вказівників, а після цього необхідно виділити пам’ять під кожен одновимірний масив. В загальному випадку виділення пам’яті під двовимірний масив включає наступні оператори:

тип ***ім’я_масиву* = new *тип** [*кількість_рядків*];
for (int i = 0; i < *кількість_рядків*; i++)
ім’я_масиву [i] = new *тип* [*кількість стовпчиків в і-му рядку*];

Вивільнення пам'яті, що відведено під двовимірний динамічний масив, виконується в порядку, зворотному процесу виділення пам’яті:

for (int i = 0; i < *кількість_рядків*; i++)
delete [] *ім’я_масиву* [i];
delete [] *ім’я_масиву*;

Приклад виділення пам’яті під матрицю розміром 4x5, яка містить дійсні елементи:

```
//оголошення двовимірного динамічного масиву 4x5 елементів:
float **a = new float* [4]; // 4 рядки в масиві
    for (int i = 0; i < 4; i++)
        a[i] = new float [5]; // і 5 стовпчиків
//a  - масив вказівників (кожен вкзівник вказує на виділену ділянку пам'яті розмір
ом в 5 дійсних чисел типу float)
```

Вивільнення пам'яті, що відведено під описаний двовимірний динамічний масив:

```
    for (int i = 0; i < 4; i++)
        delete [] a[i];
    delete [] a;
```

Використання динамічного способу виділення пам’яті дозволяє створювати зубчасті масиви – тобто в загальному випадку рядки двовимірного масиву не обов’язково повинні бути однакового розміру.

5.7 Особливості обробки файлів в C++

5.7.1 Загальна інформація про файли та потоки в C++

Файл – це конкретний пристрій введення/виведення. Потік – це узагальнений пристрій введення/виведення, який підтримує єдиний інтерфейс, що не залежить від того, до якого конкретного пристрою здійснюється доступ.

Всі потоки поводяться схожим чином. Оскільки вони не залежать від фізичних пристроїв, то та ж функція, яка виконує запис в дисковий файл, може ту ж операцію виконувати і на іншому пристрої, наприклад, на консолі.

Види потоків:

- текстові;
- двійкові (бінарні).

Текстовий потік – це послідовність символів. (розмір символу 1 байт). У стандарті С вважається, що текстовий потік організований у вигляді рядків, кожен з яких закінчується символом нового рядка. Однак в кінці останнього рядка цей символ не є обов'язковим. У текстовому потоці на вимогу базової середовища можуть відбуватися певні перетворення символів. Наприклад, символ нового рядка може бути замінений парою символів - повернення каретки і переведення рядка. Тому може і не бути однозначної відповідності між символами, які пишуться (читаються), і тими, які зберігаються в зовнішньому пристрої. Крім того, кількість тих символів, які пишуться (читаються), і тих, які зберігаються в зовнішньому пристрої, може також не збігатися завдяки можливим перетворенням.

Двійковий (бінарний) потік – це послідовність байтів, яка взаємно однозначно відповідає байтам на зовнішньому пристрої, причому ніякого перетворення символів не відбувається. Крім того, кількість тих байтів, які пишуться (читаються), і тих, які зберігаються на зовнішніх пристроях, однакова. Однак в кінці двійкового потоку може додаватися декілька нульових байтів (це визначається додатком). Такі нульові байти, наприклад, можуть використовуватися для заповнення вільного місця в блоці пам'яті незначної інформацією, щоб вона в точності заповнила сектор на диску.

У мові С (С++) файлом може бути все що завгодно, починаючи з дискового файлу і закінчуючи терміналом або принтером. Потік пов'язують з певним файлом,

виконуючи ***операцію відкриття***. Як тільки файл відкритий, можна проводити обмін інформацією між ним і програмою.

Не всі файли мають однакові можливості. Наприклад, до дискового файлу прямий доступ можливий, в той час як до деяких принтерів – ні. Таким чином, ***всі потоки однакові, а файли – ні***.

Якщо файл може підтримувати ***запити на місце розташування*** (вказівник поточної позиції), то при відкритті такого файлу вказівник поточної позиції у файлі встановлюється в початок. При читанні з файлу кожного символу (або запису в нього) вказівник поточної позиції збільшується, забезпечуючи тим самим просування по файлу.

Файл від'єднується від певного потоку (тобто розривається зв'язок між файлом і потоком) за допомогою ***операції закриття***. При закритті файлу, відкритого з метою виведення, вміст (якщо воно є) пов'язаного з ним потоку записується на зовнішній пристрій. Цей процес, який зазвичай називають дозапис потоку, гарантує, що ніяка інформація випадково не залишиться в буфері диску. Якщо програма завершує роботу нормально, тобто або main () повертає керування операційній системі, або викликається exit (), то всі файли закриваються автоматично. У разі аварійного завершення роботи програми, наприклад, в разі краху або завершення шляхом виклику abort (), файли не закриваються.

5.7.2 Операції з файлами бібліотеки stdio.h

У кожного потоку, пов'язаного з файлом, є керуюча структура, яка містить інформацію про файл; вона має тип **FILE**. Введення або виведення від кожного пристрою автоматично перетворюється системою введення/виведення в потік.

Вказівник файлу –це вказівник на структуру типу FILE. Він вказує на структуру, яка містить різні відомості про файл, наприклад, його ім'я, статус і покажчик поточної позиції в початок файлу. По суті, вказівник файлу визначає конкретний файл і використовується відповідним потоком при виконанні функцій введення / виводу. Щоб виконувати в файлах операції читання і запису, програми повинні використовувати вказівники відповідних файлів.

В таблиці 5.1 наведені основні операції в С (С++) для роботи з файлами, які описані за допомогою структури FILE.

Таблиця 5.1 – основні операції для роботи з файлами

fopen()	Відкриває файл
fclose()	Закриває файл
putc()	Записує символ в файл
fputc()	То ж, що і putc (), тільки з файлами
getc()	Читає символ з файлу

fgetc()	То ж, що і getc (), тільки з файлами
fgets()	Читає рядок з файлу
fputs()	Записує рядок в файл
fseek()	Встановлює вказівник поточної позиції на певний байт файлу
ftell()	Повертає поточне значення вказівника поточної позиції у файлі
fprintf()	Для файлу той же, що printf () для консолі
fscanf()	Для файлу той же, що scanf () для консолі
feof()	Повертає значення true (істина), якщо досягнуто кінець файлу
ferror()	Повертає значення true, якщо сталася помилка
rewind()	Встановлює вказівник поточної позиції в початок файлу
remove()	Стирає файл
fflush()	Дозапис потоку в файл

Порядок роботи з файлом, що описаний структурою FILE:

1. Оголошення змінної-вказівника файлу:

*FILE *ім'я_змінної;*

2. Відкриття файлу для виконання операцій. Спосіб відкриття залежить від майбутніх операцій.
3. Закриття файлу.

Відкриття файлу.

Функція fopen () відкриває потік і пов'язує з цим потоком певний файл. Потім вона повертає вказівник цього файлу. Найчастіше під файлом мається на увазі дисковий файл.

Прототип функції fopen ():

*FILE *fopen(const char *ім'я_файлу, const char *режим);*

Імя_файлу – це вказівник на рядок символів, що представляє собою допустиме ім'я файлу, в яке також може входити специфікація шляху до цього файлу. Рядок, на який вказує режим, визначає, яким чином файл буде відкритий.

У таблиці 5.2 показано, які значення рядка ***режим*** є допустимими. Рядки, подібні "r + b" можуть бути представлені і у вигляді "rb +".

Таблиця 5.2 – Режими відкриття файлу

R	Відкрити текстовий файл для читання

W	Створити текстовий файл для запису
A	Додати в кінець текстового файлу
Rb	Відкрити двійковий файл для читання
Wb	Створити двійковий файл для запису
Ab	Додати в кінець файлу
r+	Відкрити текстовий файл для читання / запису
w+	Створити текстовий файл для запису / читання
a+	Додати в кінець текстового файлу або створити текстовий файл для читання / запису
r+b	Відкрити двійковий файл для читання / запису
w+b	Створити двійковий файл для читання / запису
a+b	Додати в кінець файлу або створити бінарний файл для читання / запису

Функція fopen () повертає вказівник файлу. Якщо при відкритті файлу відбувається помилка, то fopen () повертає порожній (null) вказівник.

Приклад відкриття файлу:

```
FILE *fp;
fp = fopen("test", "w");
```

Або

```
FILE *fp;

if ((fp = fopen("test","w"))==NULL) {
  cout<<"File open error\n";
  exit(1);
}
```

В наведених прикладах файл з іменем test відкривається для запису (параметр "w" функції fopen). Змінна fp у випадку успішного відкриття файлу буде вказувати на перший байт файлу test.

Закриття файлу.

Функція fclose () закриває потік, який був відкритий за допомогою виклику fopen (). Функція fclose () записує в файл всі дані, які ще залишалися в дисковому буфері, і проводить, так би мовити, офіційне закриття файлу на рівні операційної системи. Відмова при закритті потоку тягне всілякі неприємності, включаючи з втрати даних, зіпсованих файлів і можливих періодичних помилок в програмі. Функція fclose () також звільняє блок управління файлом, пов'язаний з цим потоком, даючи можливість використовувати цей блок знову. Так як кількість одночасно відкритих файлів обмежена, то, можливо, доведеться закривати один файл, перш ніж відкривати інший.

Прототип функції fclose ()

*int fclose(FILE *вказівник_на_файл);*

Повернення нуля означає успішну операцію закриття. У разі ж помилки повертається EOF. Щоб точно дізнатися, в чому причина цієї помилки, можна використовувати стандартну функцію ferror (). Зазвичай відмова при виконанні fclose () відбувається тільки тоді, коли з якихось причин зникає доступ до диску (наприклад, передчасне видалення) або на диску не залишилося вільного місця.

5.7.3 Введення/виведення з використанням потоків

Для роботи з файлами через потоки використовується бібліотека потокового введення-виведення **fstream**:

- ofstream визначає потік введення;
- ifstream визначає потік виведення.

Послідовність операцій з потоками така ж сама, як і при роботі зі структурою FILE.

1. Об’явлення змінної, пов’язаної з потоком відповідного типу. Тип потоку залежить від майбутніх операцій.
2. Відкриття потоку для виконання операцій (зв’язування конкретного файлу з визначеним потоком).
3. Закриття потоку (розривання зв’язку файлу з потоком).

Відкриття потоку:

F.open(file, mode);

F – змінна, описана як ofstream, file – повне ім'я файлу на диску, mode – режим роботи з файлом. Зверніть увагу на те, що при вказанні повного імені файлу на диску потрібно ставити подвійний слеш замість одинарного.

Режими відкриття файлових потоків наведено в таблиці 5.3.

Таблиця 5.3 – Режими відкриття файлових потоків

ios :: in	відкрити файл в режимі читання даних; режим є режимом за замовчуванням для потоків ifstream
ios :: out	відкрити файл в режимі запису даних (при цьому інформація про існуючий файлі знищується); режим є режимом за замовчуванням для потоків ofstream;
ios :: app	відкрити файл в режимі запису даних в кінець файлу
ios :: ate	пересунутися в кінець вже відкритого файлу
ios :: trunc	очистити файл, це ж відбувається в режимі ios :: out
ios :: nocreate	не виконувати операцію відкриття файлу, якщо він не існує

ios :: noreplace	не відкривати існуючий файл
ios::binary	відкрити файл, як бінарний

Параметр mode може бути відсутнім, в цьому випадку файл відкривається в режимі за замовчуванням для даного потоку.

Після вдалого відкриття файлу (в будь-якому режимі) в змінній F буде зберігатися true, в іншому випадку false. Це дозволить перевірити коректність операції відкриття файлу.

Операції з потоком

Після відкриття файлу в режимі запису, в нього можна писати точно так же, як і на екран, тільки замість стандартного пристрою виведення **cout** необхідно вказати ім'я відкритого файлу.

Після відкриття файлу в режимі читання, з нього можна читати так же, як і з клавіатури, вказуючи замість стандартного пристрою виведення cin ім'я відкритого файлу.

Для визначення того, закінчився потік чи ні, використовується функція **F.eof(),** яка повертає логічне значення: true або false, залежно від того чи досягнуто кінець файлу.

Закриття потоку

Закриття потоку здійснюється за допомогою оператора:

F.close ();

Приклад 5.18. Робота з текстовим файлом. Програма створює текстовий файл, записує в нього кілька чисел і потім виводить числа з файлу через крапку з комою.

```
#include<iostream>
#include<fstream> //бібліотека для роботи с файлом

using namespace std;

void main()
{
	ifstream in; //вхідний потік
	ofstream out;//вихідний потік
	out.open("test.txt"); // відкриття файлу test.txt як вихідного потоку
	int x = 10;
	for (int i = 0; i < 5; i++)
	{
		out << x << "   "; //запис числа x у вихідний потік, числа записуют
ься через пробіл
		x += 5;
	}
	out.close();// закриття вихідного потоку
	in.open("test.txt"); // відкриття файлу test.txt як вхідного потоку
	while (!in.eof()) //обробляємо вхідний потік, доки він не скінчиться
	{
```

```
                in >> x; //зчитуємо значення з вхідного потоку
                cout << x << "; ";
        }

        in.close();// закриваємо вхідний потік
        system("pause");
}
```

Для того, щоб правильно читати дані з файлів, що зберігають дані в бінарному вигляді, потрібно знати загальну структуру зберігання всередині файлу. Коли ми говоримо про бінарні дані, тип char виступає в ролі типу byte. Щоб ми могли записати якесь значення в бінарному представленні, нам потрібно для початку вивести це бінарне представлення, а щоб записалася правильна кількість байт, потрібно явно вказувати цю кількість.

Приклад 5.19. Робота з бінарним файлом. Бінарний файл 1.txt містить ціле (int) та дійсне (double) число. Вивести вміст файлу на екран.

```
#include <fstream>      //бібліотека для роботи с файлом
#include <iostream>

using namespace std;

int main() {
   const char* FName = "C:\\1.txt";      //Шлях до файлу на диску

   int x = 0;                                    //Змінні, які виступають буфером д
ля читання з файлу
   double y = 0;

    /*Початок роботи з файлом*/
   ifstream in(FName,ios::binary); // відкриваємо файл для читання як бінарний
        in.read((char*)&x, sizeof(x));          //перенос байтів із файлу в  буфер
ну змінну x
        in.read((char*)&y, sizeof(y));          //перенос байтів із файлу в  буфер
ну змінну y
   in.close();
   /*Кінець роботи с файлом*/

    cout << x << '\n' << y << '\n';//виведення вмісту буферних змінних
    cin.get();
}
```

Для визначення кількості зчитаних/записаних байт можна використовувати конструкцію (char*)&x, sizeof(x), де x – буферна змінна.

Контрольні питання

1. Що таке масив?
2. Як оголошується та ініціюється масив у C++?
3. Які обмеження мають масиви фіксованої довжини?
4. Як здійснюється доступ до елементів масиву?
5. Що таке індекс елемента масиву?

6. Які типові помилки пов’язані з індексацією?
7. Чому необхідно контролювати межі масиву?
8. Що таке двовимірний масив?
9. Як оголошується та ініціюється двовимірний масив?
10. Як здійснюється доступ до елементів матриці?
11. У чому особливості зберігання багатовимірних масивів?
12. Як використовуються вкладені цикли для обробки матриць?
13. Що таке символьний тип даних?
14. Як подається рядок у процедурному стилі C++?
15. Яке призначення нуль-символу?
16. Як зчитуються та виводяться рядки?
17. Які обмеження мають С-рядки?
18. Що таке структура в C++?
19. Яке призначення структур?
20. Як оголошується структура?
21. Як здійснюється доступ до полів структури?
22. Що таке масив структур?
23. Чим структура відрізняється від масиву?
24. Які типи файлів використовуються у C++?
25. Чим файл відрізняється від потоку?
26. Наведіть операції для роботи з файлами та потоками.
27. Які типові помилки виникають при роботі з файлами?

6 СТРУКТУРУВАННЯ ПРОГРАМ

6.1 Функція, як елемент структурного програмування

Підпрограма є важливим елементом структурного програмування. Спочатку підпрограми з'явилися як засіб оптимізації програм за обсягом займаної пам'яті – вони дозволили не повторювати в програмі ідентичні блоки коду, а описувати їх одноразово і викликати в міру необхідності. До теперішнього часу дана функція підпрограм стала допоміжної, головне їх призначення – ***структуризація програми з метою зручності її розуміння і супроводу.***

Виділення набору дій в підпрограму і виклик її в міру необхідності дозволяє логічно виділити цілісну підзадачу, що має типове рішення. Така дія має ще одну (крім економії пам'яті) перевагу перед повторенням однотипних дій. Будь-яка зміна (виправлення помилки, оптимізація, розширення функціональності), зроблена в підпрограмі, автоматично відбивається на всіх її виклики, в той час як при дублюванні кожну зміну необхідно вносити в кожне входження змінюваного коду. Навіть в тих випадках, коли в підпрограму виділяється одноразовий набір дій, це виправдано, оскільки дозволяє скоротити розміри цілісних блоків коду, що складають програму, тобто зробити програму більш зрозумілою і доступною для огляду.

Підпрограми в C++ реалізуються у вигляді функцій. Опис функції має такий узагальнений вигляд:

```
тип_даних          ім'я_функції( список_формальних_параметрів )
{
        оператори_тіла_функції;
}
```

Список формальних параметрів у загальному випадку складається зі списку вхідних і вихідних даних. Тобто, Якщо потрібно передавати функції якісь дані, то всередині круглих дужок оголошуються параметри функції, які відокремлюються один від одного комами.

Якщо функція в явному вигляді не повертає нічого, то це описується службовим словом void. void - це тип даних, який не може зберігати будь-які дані. void ніяк по-іншому не використовується і потрібен тільки для того, щоб компілятор міг визначити тип функції:

```
void   ім'я_функції( список_формальних_параметрів)
{
        оператори_тіла_функції;
}
```

Якщо функція повертає якийсь результат, то це визначається типом, описаним в заголовку функції перед іменем функції. При цьому всередині тіла функції обов'язково повинен бути оператор return:

тип ім'я_функції(<список формальних параметрів >)
{
оператори_тіла_функції;
return значення_що_повертається;
}

Виклик функції здійснюється наступним чином:

ім'я_функції (список_фактичних_параметрів)

Способи оголошення функцій в С++:

1. В одному файлі з main:

– перед main;

– після main.

2. В окремому файлі:

– файл *.cpp;

– файл *.cpp, *h.

Якщо функція об'являється перед main, то вона описується звичайним способом – заголовок+тіло. Якщо після main, то перед main необхідно вказати прототип функції, а її реалізацію навести після функції main. Прототипом називають заголовок функції, який завершується крапкою з комою. Прототип може не містити імен параметрів, а тільки опис їх типів та способів передачі. Порядок та типи параметрів в прототипі та подальшому описі функції повинні бути однаковими.

Передача інформації у функцію здійснюється наступними способами:

– за значенням;
– через вказівник;
– через посилання.

При передачі за значенням передається копія параметру, зміна значення параметру ніяк не впливає на значення змінних у блоці, в якому функція була викликана. Синтаксис параметру-значення:

тип ім'я_параметру

Таким способом можна передавати як змінні, так і константні значення. Приклад функції, яка приймає цілочисельний параметр:

```
void f1( int a)
{
...
a=10;
…
}
```

Приклади викликів функції f1:

```
int x=3;
f1(x);
…
f1(5);
```

В жодному з випадків будь-які дії з параметром всередині функції не відобразяться на даних, які було передано у функцію.

При передачі через вказівник, у функцію передається адреса змінної або іншого об’єкту. В цьому випадку окрема пам’ять для параметру не виділяється, що корисно, наприклад, при обробці великих обсягів даних. Будь-які зміни значення параметру призводять до зміни об’єкту в блоці виклику, оскільки вони використовують спільну пам'ять. Синтаксис параметру-вказівника:

*тип * ім’я_параметру*

Через вказівники можна передавати лише адреси змінних, константні значення таким способом передавати не можна. При зверненні до значення, що зберігається за адресою відповідного параметру, в тілі функції необхідно використовувати оператор розіменування.

Приклад функції, яка приймає цілочисельний параметр-вказівник:

```
void f2( int * a)
{
...
*a=10;
…
}
```

Приклади викликів функції f2:

```
int x=3;
f2(&x); //зміна a всередині функції призведе до відповідної зміни x, оскільки і a,
і x вказують на одну й ту ж комірку пам’яті
…
f2(5);//такий виклик є синтаксично невірним!!!
…
int *p = new int (0);
f2(p);
```

При передачі через посилання & створюється нове посилання на об’єкт, що виступає параметром функції. Синтаксис параметру-посилання:

тип & ім’я_параметру

Так само, як і у випадку з вказівниками, такий параметр можна змінювати всередині функції і всі зміни будуть насправді відбуватися з об’єктом, який є фактичним параметром. Однак, на відміну від використання вказівника, параметр, переданий за посиланням можна використовувати як звичайну змінну, без застосування операції розіменування.

Приклад функції, яка приймає цілочисельний параметр-посилання:

```
void f3( int & a)
{
...
a=10;
…
}
```

Приклади викликів функції f3:

```
int x=3;
f2(x); //зміна a всередині функції призведе до відповідної зміни x
…
f2(5);//такий виклик є синтаксично невірним, оскільки неможливо оперувати адресою константи
```

Коли в якості аргументу функції використовується масив, то передається тільки адреса масиву, а не копія всього масиву. При виконанні функції з ім'ям масиву в функцію передається вказівник на перший елемент масиву. Параметр повинен мати тип, сумісний з вказівником. Існує три способи об’явлення параметра, призначеного для отримання вказівника на масив:

тип ім’я_масиву[розмір]
тип ім’я_масиву[]
*тип *ім’я_масиву*

Усі три методи оголошення параметра призводять до однакового результату – створення вказівника. Якщо масив не змінюється всередині функції, то після типу масиву вказується слово *const*.

Всі змінні, які описані в заголовку функції та в межах її тіла, доступні лице в цій функції. Поза межами функції вони недоступні. Змінні, що описані у блоці, зовнішньому до блоку виклику та блоку опису функції, доступні так само, як і звичайні змінні тіла функції. Однак такий канал обміну інформацією дуже важко контролювати, тому використання подібних змінних у сучасних підходах до програмування не вітається.

6.2 Рекурсивні функції

Визначення функцій не можуть бути вкладеними, тобто не можна всередині тіла однієї функції визначити тіло іншої. Однак, можна викликати одну функцію з іншої. У тому числі функція може викликати сама себе.

Розглянемо функцію обчислення факторіала цілого числа. Її можна реалізувати двома способами. Перший спосіб використовує ітерацію:

```
int fact(int n)
{
     int result = 1;
     for (int i = 1; i <= n; i++)
          result = result * i;
     return result;
}
```

Другий спосіб (рекурсія):

```
int fact(int n)
{
   if (n = = 1)      // факторіал 1 дорівнює 1
          return 1;
   else             // факторіал числа n дорівнює факторіалу n-1  помноженому на n
          return n * fact(n -1);
}
```

Рекурсивна функція - це функція, яка викликає саму себе (пряма рекурсія). Непряма рекурсія - коли дві або більше функцій викликають одна одну. Коли функція викликає себе, в стеку створюється копія значень її параметрів, після чого управління передається першому оператору функції. При повторному виклику процес повторюється. Базис рекурсії складають наступні положення:

– функція містить пряму або непряму рекурсію (безпосередній виклик самої себе, або виклик через іншу функцію);

– описано хоча б один нерекурсивний спосіб досягнення результату – це необхідно для того, щоб рекурсія могла завершитися;

– параметри рекурсії на кожному виклику змінюються таким чином, щоб осягнути нерекурсивного опису – це також є умовою того, що рекурсія не буде нескінченою.

Якщо хоча б одна з умов базису не виконується, то рекурсія не є коректною.

6.3 Загальні принципи структурного програмування. Заглушки функцій

Становлення і розвиток структурного програмування пов'язане з ім'ям Едсгер Дейкстри. Основні принципи структурного програмування.

Принцип 1. Слід відмовитися від використання оператора безумовного переходу goto.

Принцип 2. Будь-яка програма будується з трьох базових керуючих конструкцій: послідовність, розгалуження, цикл.

Послідовність – одноразове виконання операцій в тому порядку, в якому вони записані в тексті програми.

Розгалуження – одноразове виконання однієї з двох або більше операцій, в залежності від виконання заданої умови.

Цикл – багаторазове виконання однієї і тієї ж операції до тих пір, поки виконується задана умова (умова продовження циклу).

Принцип 3. У програмі базові керуючі конструкції можуть бути вкладені одна в одну довільним чином. Ніяких інших засобів управління послідовністю виконання операцій не передбачається.

Принцип 4. Фрагменти програми, що повторюються, можна оформити у вигляді підпрограм (функцій). Таким же чином (у вигляді підпрограм) можна оформити логічно цілісні фрагменти програми, навіть якщо вони не повторюються. У цьому випадку в тексті основної програми, замість поміщеного в підпрограму фрагмента, вставляється інструкція «Виклик підпрограми». При виконанні такої інструкції працює викликана підпрограма. Після цього триває виконання основної програми, починаючи з інструкції, наступної за командою «Виклик підпрограми».

Принцип 5. Кожну логічно завершену групу інструкцій слід оформити як блок. Блоки є основою структурного програмування.

Блок – це логічно згрупована частина вихідного коду, наприклад, набір інструкцій, записаних підряд в вихідному коді програми. Поняття блок означає, що до блоку інструкцій слід звертатися як до єдиної інструкції. Блоки служать для обмеження області видимості змінних і функцій. Блоки можуть бути порожніми або вкладеними один в іншій. Межі блоку строго визначені. Наприклад, в if-інструкції блок обмежений фігурними дужками {...}.

Принцип 6. Всі перераховані конструкції повинні мати один вхід і один вихід. Довільні керуючі конструкції (такі, як в страві спагеті) можуть мати довільну кількість входів і виходів. Обмеживши себе керуючими конструкціями з одним входом і одним виходом, ми отримуємо можливість побудови довільних алгоритмів будь-якої складності за допомогою простих і надійних механізмів.

Принцип 7. Розробка програми ведеться покроково, методом «з гори донизу» (top-down method).

Спочатку пишеться текст основної програми, в якому, замість кожного зв'язкового логічного фрагмента тексту, вставляється виклик підпрограми, яка буде виконувати цей фрагмент. Замість справжніх підпрограм в програму вставляються фіктивні частини – заглушки, які, кажучи спрощено, нічого не роблять.

Якщо говорити точніше, заглушка задовольняє вимогам інтерфейсу замінного фрагмента (модуля), але не виконує його функцій або виконує їх частково. Потім заглушки замінюються або доопрацьовуються до справжніх

повнофункціональних фрагментів (модулів) відповідно до плану програмування. На кожній стадії процесу реалізації вже створена програма повинна правильно працювати по відношенню до більш низького рівня. Отримана програма перевіряється та налагоджується.

Після того, як програміст переконається, що підпрограми викликаються в правильній послідовності (тобто загальна структура програми вірна), підпрограми-заглушки послідовно замінюються на реально працюючі, причому розробка кожної підпрограми ведеться тим же методом, що і основний програми. Розробка закінчується тоді, коли не залишиться жодної заглушки.

Така послідовність гарантує, що на кожному етапі розробки програміст одночасно має справу з доступним для огляду і зрозумілим йому безліччю фрагментів, і може бути впевнений, що загальна структура всіх вищих рівнів програми вірна.

При супроводі та внесення змін до програми з'ясовується, в які саме процедури потрібно внести зміни. Вони вносяться, не зачіпаючи частини програми, які безпосередньо не пов'язані з ними. Це дозволяє гарантувати, що при внесенні змін і виправлення помилок не вийде з ладу якась частина програми, яка перебуває в даний момент поза зоною уваги.

Приклад 6.1. Програма обліку результатів контрольної роботи студентів.

1. Описати структуру Student, яка містить:
   - прізвище студента (символьний масив);
   - номер залікової книжки;
   - оцінку за контрольну роботу.
2. Зчитати з текстового файлу список студентів.
3. Вивести на екран усіх студентів, оцінка яких не нижча за задану.
4. Обчислити середній бал групи.

Застосуємо для чорнової реалізації підхід з використанням заглушок.

Логіка основної програми включає опис безпосередньо структури та оголошення всіх необхідних функцій. Зверніть увагу на те, що специфікації функцій повинні бути прописані правильно з урахуванням їх подальшого використання. Функція main містить всі необхідні оголошення та виклики в потрібному порядку.

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

const int MAX_STUDENTS = 100;

// Опис структури студента
struct Student {
    char surname[30];
    int recordBook;
    int mark;
};
```

```
// Оголошення функцій
int readFromFile(Student students[]);
void printStudents(const Student students[], int count);
void printStudentsByMark(const Student students[], int count, int minMark);
double calculateAverageMark(const Student students[], int count);

int main() {
    Student group[MAX_STUDENTS];
    int count;

    // Зчитування даних з файлу
    count = readFromFile(group);

    if (count == 0) {
        cout << "Дані відсутні." << endl;
        return 0;
    }

    // Виведення всіх студентів
    printStudents(group, count);

    // Виведення студентів з оцінкою не нижче 80
    printStudentsByMark(group, count, 80);

    // Обчислення та виведення середнього бала
    double average = calculateAverageMark(group, count);
    cout << "Середній бал групи: " << average << endl;

    return 0;
}
```

Реалізація функцій-заглушок може виглядати наступним чином:

```
int readFromFile(Student students[]) {
    cout << "[ЗАГЛУШКА] Виклик функції readFromFile()" << endl;
    cout << "Імітація зчитування даних з файлу." << endl;

    // Поки що дані не зчитуються
    return 0;
}

void printStudents(const Student students[], int count) {
    cout << "[ЗАГЛУШКА] Виклик функції printStudents()" << endl;
    cout << "Кількість студентів для виведення: " << count << endl;
}

void printStudentsByMark(const Student students[], int count, int minMark) {
    cout << "[ЗАГЛУШКА] Виклик функції printStudentsByMark()" << endl;
    cout << "Мінімальна оцінка для відбору: " << minMark << endl;
    cout << "Кількість студентів у масиві: " << count << endl;
}

double calculateAverageMark(const Student students[], int count) {
    cout << "[ЗАГЛУШКА] Виклик функції calculateAverageMark()" << endl;
    cout << "Обчислення середнього бала для " << count << " студентів." << endl;

    // Поки що середній бал не обчислюється
    return 0.0;
}
```

Кожна функція виводить повідомлення про свій виклик, що дозволяє переконатися, що функція підключена правильно, перевірити порядок виконання програми, виявити помилки у викликах або передачі параметрів.

Такий підхід дає змогу компілювати та запускати програму ще до реалізації основної логіки, поступово замінювати заглушки реальним кодом без зміни структури програми та використовувати програму як основу для поетапної роботи. Діагностичні повідомлення є тимчасовими і мають бути вилучені або замінені реальною функціональністю після завершення реалізації.

Однак наведений варіант заглушок не дозволяє перевірити роботу всіх функцій, оскільки не імітує зчитування з файлу, а завжди повертає в основну програму кількість зчитаних записів=0. Для таких випадків, коли заглушка по суті блокує виклики потрібних функцій, необхідно згенерувати певні тестові дані, які на етапі реалізації в подальшому повинні бути замінені на реальні дані.

Виправлений варіант заглушки у функції readFromFile імітує зчитування з файлу, заповнює масив структур наперед визначеними тестовими значеннями та повертає кількість студентів, що дозволяє виконати інші функції програми.

```
int readFromFile(Student students[]) {
    cout << "[ЗАГЛУШКА] Виклик функції readFromFile()" << endl;
    cout << "Імітація зчитування даних з файлу (тестові дані)." << endl;

    // Ініціалізація тестових даних
    students[0] = {"Koval",     10101, 85};
    students[1] = {"Shevchuk",  10102, 72};
    students[2] = {"Bondarenko",10103, 90};
    students[3] = {"Melnyk",    10104, 64};

    int count = 4;

    cout << "Зчитано студентів: " << count << endl;

    return count;
}
```

Слід врахувати, що деякі автори зазначають, що принципи структурного програмування в рівній мірі можуть застосовуватися при розробці програм як «з гори донизу», так і «знизу догори».

Дотримання принципів структурного програмування робить тексти програм, навіть досить великих, такими, що нормально читаються. Серйозно полегшується розуміння програм, з'являється можливість розробки програм в нормальному промисловому режимі, коли програму може без особливих труднощів зрозуміти не тільки її автор, а й інші програмісти. Це дозволяє розробляти досить великі для того часу програмні комплекси силами колективів розробників, і супроводжувати ці комплекси протягом багатьох років, навіть в умовах неминучих змін в складі персоналу:

1. Структурне програмування дозволяє значно скоротити число варіантів побудови програми по одній і тій же специфікації, що значно знижує складність програми і, що ще важливіше, полегшує розуміння її іншими розробниками.

2. У структурованих програмах логічно пов'язані оператори перебувають візуально ближче, а слабо пов'язані – далі, що дозволяє обходитися без блок-схем та інших графічних форм зображення алгоритмів (по суті, сама програма є власною блок-схемою).

3. Сильно спрощується процес тестування і налагодження структурованих програм.

Контрольні питання

1. Що таке функція?
2. Яке призначення функцій у програмі?
3. Як оголошується та викликається функція?
4. Що таке параметри функції?
5. Чим відрізняється передавання за значенням, через покажчик і за посиланням?
6. Яке призначення оператору return?
7. Що таке рекурсивна функція? Які види рекурсії бувають в C++?
8. Опишіть правила організації рекурсивних обчислень.
9. Що таке заглушка функції?
10. Чому модульна структура програми є важливою?

СПИСОК ЛІТЕРАТУРИ

1. Вступ до програмування мовою С++. Організація обчислень : навч. посіб. / Ю. А. Бєлов, Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставровський. – К. : Видавничо-поліграфічний центр "Київський університет", 2012. – 175 с. с.: іл. ISBN (укр.)
2. Основи програмування на C/C++ в прикладах. Частина 1: навч.-метод. посібник / Соболь М.О., Любченко Н.Ю, Паржин Ю.В., Пугачов Р.В. – Харків : НТУ "ХПІ", 2021. – 113 с.
3. Браян В. Керніган, Деніс М. Річі. Мова програмування С. Переклад: Віталій Цибуляк https://titkov.ho.ua/_KPI_/1K/BOOKS/kernigan_richi_C_UKR.pdf
4. Microsoft C++, C, and Assembler documentation. https://learn.microsoft.com/en-us/cpp/?view=msvc-170
5. Інформатика. Основи програмування та алгоритми. Мова програмування С. Лабораторний практикум [Електронний ресурс] : навчальний посібник для здобувачів ступеня бакалавра за освітніми програмами «Інтелектуальні технології радіоелектронної техніки», «Інформаційна та комунікаційна радіоінженерія», «Радіотехнічні комп’ютеризовані системи», «Інформаційне забезпечення робототехнічних систем» спеціальності 172 Телекомунікації та радіотехніки 126 Інформаційні системи та технології / КПІ ім. Ігоря Сікорського ; уклад. С. В. Вишневий, П. Ю. Катін, Є. В. Крилов. – Електронні текстові дані (1 файл: 3,3 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 221 с. – Назва з екрана. https://ela.kpi.ua/items/d0e5e7d7-a5c3-4eb5-81d8-11639218fa0b
6. Горчинський С., Борисов Д. Обґрунтування вибору мови програмування для початкових курсів програмування // Вісник Національного університету «Чернігівський колегіум» імені Т. Г. Шевченка. Вип. 24 (180) / Національний університет «Чернігівський колегіум» імені Т. Г. Шевченка ; голов. ред. М. О. Носко. Чернігів : НУЧК, 2023. С. 100-108. (Серія: Педагогічні науки)
7. Програмування: підручник [Електронний ресурс] / укладач Л. Я. Глинчук, Т.О. Гришанович; ВНУ ім. Лесі Українки. – Електронні текстові данні (1 файл: 3 201 КБ). – Луцьк: ВНУ ім. Лесі Українки, 2022. – 160 с. https://evnuir.vnu.edu.ua/handle/123456789/20649
8. Карпенко Н. В., Герасимов В. В. Сучасний підхід до програмування на мові С від нульового до просунутого рівня : навч. посіб. / Н. В. Карпенко, В. В. Герасимов — Д.: Ліра, 2022. — 418 с. https://www.researchgate.net/profile/N-Karpenko/publication/364720175_Sucasnij_pidhid_do_programuvanna_na_m

ovi_S_vid_nulovogo_do_prosunutogo_rivna/links/6358246296e83c26eb529fc4/Sucasnij-pidhid-do-programuvanna-na-movi-S-vid-nulovogo-do-prosunutogo-rivna.pdf
9. Зеленський О.С., Лисенко В.С. Основи програмування на С++ Навчальний посібник – Кривий Ріг: Державний університет економіки і технологій, 2023.-269 с.
10. Основи програмування. Лабораторний практикум для студентів денної та заочної форм навчання освітньої програми «Інженерія програмного забезпечення Інтернету речей» першого (бакалаврського) рівня вищої освіти за спеціальністю 121 Інженерія програмного забезпечення галузі знань 12 Інформаційні технології / уклад. О. В. Шпортько. Рівне: ПВНЗ "МЕГУ ім. акад. С. Дем'янчука", 2022. 104 с. https://comp.ucoz.net/AP/MProgIPZ22.pdf
11. Програмування та алгоритмічні мови. Частина 2. Програмування. Конспект лекцій.//Укладач: І. В. Назарчук. Електронне мережне навчальне видання. Київ. КПІ ім. Ігоря Сікорського, 2022. 143 с.

ЗАДАЧІ ДЛЯ САМОСТІЙНОГО ОПРАЦЮВАННЯ

Задача «Браковані тортики»

За один цикл роботи автоматизованій лінії з виробництва тортів виготовляється N тортів за однією і тою ж технологічною картою. Нормативна вага торту* відповідно до технологічної карти складає W. На виготовлення одного торту конвеєром витрачається не більше, ніж T хвилин ** (конкретний час виготовлення торту може відрізнятись та залежить від показників датчиків конвеєру, які фіксують завершення кожної стадії процесу).

Перед пакуванням торту виконується зважування виробу. Реальна вага торту може відрізнятися від нормативної. Максимально допустиме відхилення від нормативної ваги складає ±D**. Якщо в результаті зважування виявлено відхилення, що перевищує максимально допустиме, такий торт вважається браком.

1. Визначити загальну довжину робочого циклу роботи лінії для виготовлення N тортів та вивести у форматі «Г годин X хвилин».
2. Визначити загальну кількість бракованих тортів та розрахувати середню вагу тих тортів, що виявилися небракованими.
3. Визначити, в яку половину циклу роботи лінії (в першу чи другу) було більше браку.

** Одиницю вимірювання ваги визначити на власний розсуд.*

*** Зверніть увагу на обмеження предметної галузі, що можуть накладатися на вхідні дані задачі (типи даних, діапазони допустимих даних)*

****Задачу реалізувати у двох варіантах – із використанням одновимірного масиву та без нього!*

Задача «Кредит»

Людина взяла кредит у банку на суму X грн. Кожного місяця людина сплачує деяку суму виходячи із своїх фінансових можливостей. Щомісячна плата, встановлена банком, становить не менше 5% від залишкової суми кредиту. Якщо людина сплачує менше, то сума кредиту збільшується на 0.05% від залишку кредиту. Інтерфейс програми повинен передбачати наступне: введення з клавіатури загальної суми X, введення чергової суми виплат, після кожного введення чергової суми необхідно вивести, скільки людині ще залишилося виплатити (додатково можна вивести повідомлення про те, що сума збільшилася, якщо людина заплатила менше 5% від залишкової суми кредиту).

Визначити:

За скільки місяців людина виплатила кредит
Номер та суму мінімального платежу
Номер та суму максимального платежу

**Задачу реалізувати у двох варіантах – із використанням одновимірного масиву та без нього!*

Тестовий приклад:
Input X: 1000

Input 1 payment: 100
Your debt is 900
The next minimum payment is 45

Input 2 payment: 20
You paid less than the minimum payment. The new debt is 924.
The next minimum payment is 46,2

Input 3 payment: 100
Your debt is 824
The next minimum payment is 41,2

Input 4 payment: 830
Your debt is 0

6 UAH in excess of the loan will be returned upon your request

Loan repaid in 4 payments

The minimum payment is 20 UAH (2 payment)
The maximum payment is 830 UAH (4 payment)

Задача «Хто в домі транжира»

Марічка і Антон посварилися через те, що їм постійно не вистачає грошей. Щоб з'ясувати, хто в цьому винен, вони вирішили N місяців рахувати сімейний бюджет.

Кожного із звітних тижнів вони фіксували, хто скільки грошей заробив, і хто скільки грошей витратив. Якщо в кінці тижня в когось залишалися гроші, то людина клала їх у ***власну скарбничку***. Гроші із скарбнички брати до кінця експерименту брати не можна було. Якщо комусь протягом тижня не вистачало грошей, то можна було нестачу позичити у батьків.

Вхідними даними задачі є: інформація про щотижневі витрати кожного, інформація про щотижневі добутки кожного, інформація про щотижневі позичання грошей у батьків.

Визначити:

1. Скільки грошей в кожного накопичилося в скарбниці за N тижнів.
2. Хто більший транжира.
3. Чий середній баланс кращій.
4. Кому довелося хоча б раз позичати гроші у батьків.

**При виконанні роботи використовувати одновимірні масиви*

Задача «Нестача на складі»

На склад поступила партія з N пакунків з цукерками. В результаті ревізії виявилося, що з деяких пакунків при транспортуванні зникла частина товару.

1. Визначити вагу нестачі в кожному пакунку, якщо відомі їх необхідне та реальне нетто.
2. Визначити номери пакунків, в яких виявлено нестачу.
3. Розрахувати загальну вагу нестачі.

** Одиницю вимірювання ваги визначити на власний розсуд.*

***При виконанні роботи використовувати одновимірні масиви*

Задача «Все переплутано»

Годування на фермі через автоматизований кормороздатник організовано двічі на день. План годування складається на місяць та включає за кожен день 2 числа: вагу кормів на ранок, вагу кормів на вечір (в кілограмах).

При формуванні плану оператор допустив помилку та переплутав вечірню та денну норму. Скласти програму, яка виправляє план, міняючи місцями норми на кожен день.

Тестовий приклад (фрагмент плану на 3 дні):

Вхідний план: 10, 7, 12, 9, 12, 10, 15, 13

Відкоригований план: 7, 10, 9, 12, 10, 12, 13, 15

**При виконанні роботи використовувати одновимірні масиви*

Задача «Труба»

Автоматична лінія виробляє поліетиленову трубу у відрізках. Виробництво ведеться партіями. На 1м труби витрачається X грам сировини. Для організації роботи над однією партією труби треба підготувати такі дані:

- кількість відрізків в партії;
- довжини відрізків, що повинні бути виготовлені (технологічне обмеження максимальної довжини відрізку складає 12 м, мінімальної довжини відрізку – 50 см);
- максимальну та мінімальну довжину відрізку в партії (дані необхідні для визначення позицій обмежувачів різаків автоматичної лінії);
- розрахунок обсягів сировини, необхідної для виготовлення всієї партії.

Програма повинна забезпечити введення необхідних даних з клавіатури, формування масиву довжин відрізків і інші показники, вказані в завданні.

Примітка – в задачі використовувати динамічний масив для представлення довжин відрізків.

**** Додаткове завдання:***

Сформувати дані для виготовлення N партій труби (із використанням зубчастих динамічних масивів).

Задача «Сам собі фотограф»

Для представлення кольорових зображень часто використовується формат RGB (Red, Green, Blue). Одному пікселю відповідає 3 числа, кожне з яких містить значення відповідного кольору. Самі значення кодуються числами від 0 до 255. Створіть 3 матриці що містять кольори пікселів зображення розміром NxM (кольори можна генерувати випадковими числами) та виконайте наступні перетворення зображення:

1) отримайте зображення у відтінках сірого. Для цього необхідно розрахувати яскравість кожного пікселу. Один із методів розрахунку яскравості визначається формулою Y=0.2126R+ 0.7152G+0.0722B
2) зменшіть шум на вхідному зображенні за допомогою медіанного фільтру. Медіанний фільтр може бути реалізований у вигляді вікна розміром 3x3 пікселі, яке послідовно переміщується по зображенню. Значення пікселів у вікні сортуються за зростанням та обчислюється значення, що стоїть посередині – воно є результатом фільтру (медіаною). Центральний піксель у вікні замінюється на отриману медіану. Медіанний фільтр застосовується для зображення у відтінках сірого.

Задача «Шифрування»

Першим завданням в школі юного шифрувальника було розробити свій власний шифр. Треба написати програму (чи декілька програм, кожна для свого способу шифрування), яка дозволяє зашифрувати повідомлення вказаними способами та розшифрувати їх.

На вхід програма отримує повідомлення довжиною не більше 100 елементів та необхідні параметри шифрування (якщо вони є), потім на екран виводиться зашифроване повідомлення.

Способи шифрування, які запропонували юні шифрувальники:

1. Кожний елемент вхідного повідомлення замінюється на його різницю із першим елементом (віднімаємо значення першого елементу від поточного, перший елемент не змінюється).
2. Зашифроване повідомлення утворюється віддзеркаленням відносно середини вхідного повідомлення.
3. Для повідомлення довжиною N задається ключ, який містить не більше ніж N чисел. Всі числа в ключі повинні бути різними. Кожне число ключа визначає номер елементу у вхідному повідомленні, який треба подвоїти.
4. Для кожного елементу вхідного повідомлення записується наступне: сам елемент, скільки разів він зустрічається в повідомленні та позиції, на яких він зустрічається.

Зауваження стосовно реалізації задачі.

Вхідне повідомлення може мати різні представлення: набір цілих чисел; набір символів (рядок). Якщо зможете, реалізуйте обидва варіанти.

Результуюче повідомлення може мати числове чи символьне представлення. Спосіб представлення визначте самостійно з урахуванням особливостей способу шифрування.

**Додаткове завдання – виконати розшифрування зашифрованих повідомлень.*

Задача «Точки»

Задано множину із N точок в двовимірному просторі. Визначити:

1) Номер та координати точки, що розташована найближче до початку координат. Якщо таких точок декілька, то вивести параметри всіх.
2) Довжину ломаної лінії, яку можна утворити, поєднавши послідовно всі точки множини.

Вимоги до програми:

- Використовувати для представлення точки структуру struct.
- Для вирішення задач використовувати функції.

Задача «Метеостанція»

Метеостанція міста X збирає дані з зовнішніх датчиків та пристроїв і передає їх в головний офіс. Дані фіксуються через кожні пів години та записуються у текстовий файл. Кожен запис починається з нового рядка та містить наступну інформацію:

<температура> <вологість повітря> <швидкість вітру> <атмосферний тиск> <тип опадів>

Дані розділяються пробільними символами (пробіл, табуляція).

Приклад рядків файлу:

1 75 3,5 747 0
-1 77 3 748 -1

Температура визначається цілим числом (можуть зустрічатися від'ємні числа), вологість повітря визначається цілим числом від 0 до 100 (відсоток вологості), швидкість вітру – дійсне число (м/с), атмосферний тиск – ціле число. Тип опадів кодується цілим числом в діапазоні від -3 до 5 з наступними розшифровками:

0 без опадів (no precipitation)
-1 слабкий сніг (light snow)
-2 сніг (snow)
-3 сильний сніг (snowfall)
1 морось (drizzle)
2 слабкий дощ (light rain)
3 дощ (rain)
4 злива (rainfall)
5 туман (fog)

Примітка: При виконанні основного і додаткового завдань можна використовувати структури struct та проміжні бінарні файли. **Використання великих масивів, як проміжних буферів для зберігання даних вхідних файлів, не вітається!**

ЗАВДАННЯ

Обов'язкова частина:

Створити в будь-якому текстовому редакторі файл з декількома даними метеостанції згідно з описаним форматом. Кількість рядків даних не менше 5.

Скласти програму, яка робить наступні дії:

1. Зчитує з існуючого текстового файлу дані та виводить їх на екран у вигляді:

   t=температура; w=вологість повітря; s=швидкість вітру; p=атмосферний тиск; pr=тип опадів

   Приклад виведення:

   t=1; w=75; s=3,5; p=747; pr=no precipitation

   t=-1; w=77; s=3; p=748; pr=light snow
2. Розраховує за даними з файлу наступні показники:
   - кількість проміжків часу, врахованих у файлі;
   - середню температуру повітря;
   - мінімальну температуру повітря;
   - максимальну температуру повітря;
   - кількість проміжків часу, коли спостерігався вказаний вид опадів (користувач вводить з клавіатури код типу опадів).
3. Виводить розраховані дані на екран та у вихідний текстовий файл (формат виведення та структура вихідного файлу – на розсуд студента).

Примітка: імена вхідного та вихідного файлів задати в програмі константними, обробку помилок при роботі з файлами можна не реалізовувати.

Додаткове завдання (можна реалізувати частково):

1. Передбачити в програмі можливість введення імен вхідного та вихідного файлу з клавіатури.
2. Передбачити в програмі обробку помилок при роботі з файлами:
   - неможливість відкрити вхідний файл;
   - неможливість створити вихідний файл;
   - помилки у вхідному файлі (невідповідність даних формату, некоректні дані з точки зору обмежень, зайві дані, недостатньо даних тощо);
3. Реалізувати в програмі можливість додавати дані у вхідний файл.
4. Розрахувати та вивести на екран і в інший вихідний файл дані, необхідні для побудови по обраному показнику гістограми. Якщо обробляються декілька показників, то кожному з них повинен відповідати свій файл.

Гістограма – це графічне відображення статистичних даних у вигляді стовпчастої діаграми. Дані для побудови гістограми представляють собою пари чисел:

значення показника - кількість даних, що відповідають цьому показнику

Гістограму можна будувати для окремих точкових значень. Наприклад, якщо є наступний набір температур:

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

то, із використанням окремих точкових значень, дані для гістограми можна побудувати наступним чином:

1) Сортуємо числа в порядку їх зростання

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

2) Визначаємо для кожного числа, скільки раз воно зустрічається в наборі.

   Результат:

   -2 2
   -1 4
   0 4
   1 5
   2 1
   3 2
   4 3

При побудові гістограми для діапазонів даних, порядок розрахунків може бути наступний (вхідний набір такий самий, як і для попереднього варіанту розрахунку).

1) Визначаємо мінімальне та максимальне значення в наборі даних. Для нашого прикладу min=-2, max=4.
2) Задаємо кількість діапазонів, наприклад n=4.
3) Розраховуємо довжину діапазону d= (max-min)/n. Для нашого випадку d= 1,5.
4) Визначаємо кордони діапазонів та рахуємо, скільки значень потрапило в кожен діапазон.

   Результат:

   [-2;-0,5) 6
   [-0,5;1) 4
   [1;2,5) 6
   [2,5;4] 5