Міністерство освіти і науки України
Міжнародний гуманітарний університет
Кафедра комп’ютерних наук

Русу О.П.

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

Конспект лекцій
для здобувачів вищої освіти,
які навчаються за спеціальностями
галузі «Інформаційні технології»

Одеса 2025

Затверджено Вченою Радою Міжнародного гуманітарного університету.
Протокол № 5 від 20 січня 2025 р.

Русу О.П. Алгоритмізація та програмування. Конспект лекцій для здобувачів вищої освіти, які навчаються за спеціальностями галузі «Інформаційні технології» [Електронне видання]. Кафедра комп'ютерних наук Міжнародного гуманітарного університету. Одеса, 2025. 43 с.

Навчальна дисципліна «Алгоритмізація та програмування» є базовою складовою професійної підготовки фахівців зі спеціальності «Інженерія програмного забезпечення» та формує фундаментальні знання з алгоритмічного мислення і принципів розробки програмного забезпечення. У межах дисципліни вивчаються поняття алгоритму, його властивості та способи подання, базові алгоритмічні конструкції, типи даних, функції та масиви, а також принципи модульної побудови програм. Особлива увага приділяється формалізації задач, реалізації алгоритмів засобами мови програмування високого рівня та аналізу коректності й ефективності програмних рішень. Дисципліна закладає теоретичну та практичну основу для подальшого опанування фахових курсів і формує базові компетентності розробника програмного забезпечення.

ЗМІСТ

ТЕМА 1. АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ В ЖИТТЄВОМУ ЦИКЛІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Основні поняття алгоритмізації та способи подання алгоритмів

Алгоритмізація є фундаментальним етапом створення програмного забезпечення. Будь-яка програмна система починається з формулювання задачі та побудови алгоритму її розв'язання. Саме алгоритм визначає послідовність дій, які необхідно виконати для отримання потрібного результату. Тому розуміння поняття алгоритму, його властивостей, а також способів подання алгоритмів є базовою передумовою формування навичок програмування.

Поняття алгоритму виникло задовго до появи комп'ютерів. У найзагальнішому вигляді алгоритм можна визначити як точний і зрозумілий опис послідовності дій, виконання яких приводить до розв'язання певної задачі. У сучасній інформатиці алгоритм розглядається як формалізований набір інструкцій, які може виконати певний виконавець. Ці інструкції повинні бути чітко визначені, мати однозначне трактування та виконуватися у визначеній послідовності.

Термін алгоритм походить від латинізованого імені середньоазійського математика Аль-Хорезмі, який у IX столітті описав правила виконання арифметичних обчислень. З часом поняття алгоритму стало основою математичної теорії обчислень, а пізніше — ключовим поняттям комп'ютерних наук.

Алгоритм має низку властивостей, які відрізняють його від довільного опису дій. До основних властивостей алгоритму належать визначеність, дискретність, масовість та результативність.

Визначеність означає, що кожна команда алгоритму повинна бути сформульована однозначно і не допускати різних тлумачень. Виконавець алгоритму повинен точно розуміти, яку саме дію необхідно виконати на кожному етапі. Якщо команда сформульована нечітко або допускає неоднозначне трактування, виконання алгоритму може призвести до неправильного результату або взагалі стати неможливим. Наприклад, інструкція на кшталт «обробити дані» не є визначеною, оскільки не пояснює, які саме дії потрібно виконати. Натомість команда «додати два числа та записати результат у змінну» має однозначний зміст.

Дискретність означає, що алгоритм складається з окремих етапів або кроків. Кожен крок виконується після завершення попереднього. Таким чином, процес розв'язання задачі розбивається на послідовність елементарних операцій. Саме така структура дає змогу виконувати алгоритм автоматизовано, оскільки комп'ютер виконує команди одну за одною у визначеному порядку.

Масовість означає, що алгоритм призначений для розв'язання не однієї конкретної задачі, а певного класу однотипних задач. Наприклад, алгоритм знаходження суми двох чисел може застосовуватися для будь-яких значень цих чисел. У цьому полягає універсальність алгоритмічного підходу: один і той самий алгоритм може використовуватися для обробки різних вхідних даних.

Результативність означає, що виконання алгоритму повинно завершуватися отриманням результату за скінченну кількість кроків. Якщо алгоритм не завершується або не

приводить до певного результату, він не може вважатися коректним. Ця властивість гарантує, що алгоритм не містить нескінченних циклів без умов завершення.

Важливим поняттям у теорії алгоритмів є поняття виконавця алгоритму. Виконавець — це об'єкт або система, здатна виконувати команди алгоритму. У ролі виконавця може виступати людина, комп'ютер, робот або інша технічна система. Основною характеристикою виконавця є набір дій, які він може виконувати. Цей набір називають системою команд виконавця.

Алгоритм повинен бути сформульований у термінах, зрозумілих виконавцю. Наприклад, алгоритм, призначений для комп'ютера, має бути описаний мовою програмування або іншою формальною системою запису команд. Якщо ж алгоритм виконує людина, інструкції можуть бути подані у вигляді текстового опису.

Кожен виконавець має обмежений набір можливих операцій. Наприклад, комп'ютер може виконувати арифметичні операції, операції порівняння, операції читання та запису даних, а також керувати послідовністю виконання команд. Саме на основі цих елементарних операцій будуються складні алгоритми.

Перед побудовою алгоритму необхідно виконати формалізацію прикладної задачі. Формалізація — це процес перетворення реальної задачі на формальну модель, яку можна описати математично або алгоритмічно. У процесі формалізації визначаються вхідні дані задачі, необхідні обчислення та очікуваний результат.

На цьому етапі також визначаються обмеження задачі, допустимі значення параметрів та умови виконання алгоритму. Формалізація дає змогу чітко сформулювати задачу та уникнути неоднозначностей у її трактуванні. Наприклад, задача обчислення середнього арифметичного набору чисел може бути формалізована як обчислення суми всіх елементів і поділ отриманого результату на їх кількість.

Після формалізації задачі виконується побудова алгоритму її розв'язання. Для цього використовують різні способи подання алгоритмів. Найбільш поширеними є словесний опис, блок-схеми та псевдокод.

Словесний опис алгоритму є найпростішим способом подання послідовності дій. У цьому випадку алгоритм описується природною мовою у вигляді послідовності інструкцій. Такий спосіб є зрозумілим для людини, однак має певні недоліки. Природна мова може допускати неоднозначності, тому складні алгоритми важко описувати лише текстом.

Попри це, словесний опис часто використовується на початковому етапі розробки алгоритму, коли необхідно сформулювати загальну ідею розв'язання задачі. Наприклад, алгоритм знаходження найбільшого з двох чисел може бути описаний таким чином: отримати два числа, порівняти їх між собою, визначити більше число та вивести результат.

Більш формалізованим способом подання алгоритмів є використання блок-схем. Блок-схема являє собою графічне представлення алгоритму у вигляді спеціальних символів, з'єднаних лініями. Кожен символ відповідає певній операції або етапу виконання алгоритму.

У блок-схемах використовуються стандартні графічні позначення. Прямокутник використовується для позначення операції або обчислення. Паралелограм використовується для введення або виведення даних. Ромб застосовується для позначення перевірки умови та розгалуження алгоритму. Овал використовується для позначення початку або завершення алгоритму. Стрілки показують напрямок виконання алгоритму.

Блок-схеми дають змогу наочно відобразити логіку роботи алгоритму. Вони широко використовуються під час аналізу та документування алгоритмів, а також на етапі навчання

програмуванню. Візуальне представлення допомагає краще зрозуміти структуру алгоритму та взаємозв’язок між його елементами.

Ще одним поширеним способом подання алгоритмів є псевдокод. Псевдокод являє собою формалізований текстовий опис алгоритму, який нагадує програмний код, але не прив’язаний до конкретної мови програмування. У псевдокоді використовуються умовні конструкції, цикли та операції присвоєння, подібні до тих, що застосовуються у мовах програмування.

Основною перевагою псевдокоду є поєднання зрозумілості та формальності. З одного боку, псевдокод легко читається людиною, а з іншого — має достатню структурованість для подальшого перетворення на програмний код. Тому цей спосіб часто використовується під час проєктування програмних систем.

Незалежно від способу подання алгоритму, його структура будується на основі базових алгоритмічних конструкцій. У теорії алгоритмів доведено, що будь-який алгоритм можна побудувати, використовуючи три основні структури: послідовність, розгалуження та повторення.

Послідовність є найпростішою структурою алгоритму. У цьому випадку команди виконуються одна за одною у визначеному порядку. Кожна наступна операція починається лише після завершення попередньої. Прикладом послідовної структури може бути алгоритм обчислення значення математичного виразу, який складається з кількох арифметичних операцій.

Розгалуження використовується у випадках, коли подальший хід виконання алгоритму залежить від певної умови. Якщо умова виконується, алгоритм переходить до однієї гілки виконання, а якщо ні — до іншої. Така структура використовується, наприклад, під час перевірки коректності введених даних або під час вибору способу обчислення залежно від значення параметра.

Повторення використовується у випадках, коли певну послідовність дій необхідно виконати кілька разів. Повторення реалізується за допомогою циклів. Цикл може виконуватися визначену кількість разів або до тих пір, поки виконується певна умова. Ця структура широко використовується під час обробки масивів даних, виконання ітераційних обчислень або перебору варіантів.

Поєднання цих трьох базових структур дає змогу будувати алгоритми будь-якої складності. Саме на їх основі формуються програмні конструкції у мовах програмування.

Таким чином, алгоритмізація є ключовим етапом розробки програмних систем. Алгоритм визначає логіку розв’язання задачі, а його властивості забезпечують можливість автоматизованого виконання. Формалізація прикладної задачі дає змогу чітко визначити вхідні дані, необхідні обчислення та очікуваний результат. Для подання алгоритмів використовуються різні способи, серед яких словесний опис, блок-схеми та псевдокод. Незалежно від способу подання, будь-який алгоритм будується на основі базових структур: послідовності, розгалуження та повторення. Розуміння цих принципів є основою подальшого вивчення програмування та розробки програмного забезпечення.

1.2 Етапи розробки програмного забезпечення та життєвий цикл програмних систем

Створення програмного забезпечення є складним інженерним процесом, який потребує системного підходу, чіткої організації роботи та використання спеціалізованих

методів і технологій. Розробка програмних систем передбачає послідовне виконання низки взаємопов'язаних етапів, кожен з яких має власні завдання та результати. Сукупність цих етапів формує життєвий цикл програмного забезпечення.

У загальному випадку процес створення програмного продукту можна подати як послідовність таких етапів: формування вимог до програмної системи, проєктування алгоритмів та архітектури, реалізація програмного коду, тестування програмного забезпечення та його подальший супровід. Кожен із цих етапів має важливе значення для забезпечення якості програмної системи та ефективності її подальшого використання.

Першим етапом розробки програмного забезпечення є формування вимог. На цьому етапі визначаються функції програмної системи, її призначення, обмеження та умови експлуатації. Вимоги до програмного забезпечення формуються на основі аналізу потреб користувачів або замовника системи.

Процес формування вимог передбачає збір, аналіз і документування інформації про майбутню систему. На цьому етапі визначаються задачі, які має розв'язувати програмний продукт, а також способи взаємодії користувача із системою. У результаті формується перелік функціональних та нефункціональних вимог.

Функціональні вимоги описують, які саме дії повинна виконувати програмна система. Вони визначають основні функції програмного продукту, способи обробки даних та взаємодію з користувачем. Наприклад, для системи обліку даних функціональними вимогами можуть бути введення інформації, її збереження, пошук, редагування та формування звітів.

Нефункціональні вимоги визначають характеристики якості програмного забезпечення. До них належать вимоги щодо продуктивності системи, надійності, безпеки, зручності використання, сумісності з іншими програмними продуктами та апаратними засобами. Нефункціональні вимоги часто визначають технічні обмеження, у межах яких повинна працювати система.

Важливою частиною формування вимог є їх документування. Для цього створюється спеціальний документ, який містить опис функціональних можливостей системи, вимоги до її роботи та інші технічні параметри. Такий документ часто називають специфікацією вимог до програмного забезпечення.

Після визначення вимог розпочинається етап проєктування програмної системи. Проєктування передбачає розробку структури майбутнього програмного продукту, визначення його основних компонентів та способів їх взаємодії.

На цьому етапі виконується аналіз задачі та побудова алгоритмів її розв'язання. Проєктування алгоритмів є одним із ключових етапів створення програмного забезпечення, оскільки саме алгоритми визначають логіку роботи майбутньої програми.

Алгоритм повинен забезпечувати правильне та ефективне розв'язання задачі. Під час проєктування алгоритмів розробник визначає послідовність обчислень, структуру обробки даних та умови переходу між різними етапами виконання програми.

На цьому етапі також формується архітектура програмної системи. Архітектура визначає структуру програмного продукту, його основні модулі та взаємозв'язки між ними. Добре спроєктована архітектура забезпечує зручність подальшої розробки, модифікації та супроводу програмного забезпечення.

Під час проєктування широко використовуються різні методи представлення алгоритмів та структури програмної системи. До них належать блок-схеми, псевдокод,

діаграми модулів та інші засоби моделювання. Такі інструменти дають змогу описати логіку роботи системи ще до початку написання програмного коду.

Наступним етапом є реалізація програмного коду. На цьому етапі розробники створюють програму мовою програмування відповідно до алгоритмів та архітектури, визначених на попередньому етапі.

Реалізація програмного коду передбачає написання програмних модулів, організацію структури даних, створення функцій і процедур, а також реалізацію взаємодії між різними компонентами програмної системи. Під час написання коду розробники повинні дотримуватися правил програмування, використовувати зрозумілі імена змінних і функцій та забезпечувати логічну структуру програми.

Якість програмного коду має важливе значення для подальшої експлуатації програмного забезпечення. Чітка структура коду, коментарі та дотримання стандартів програмування полегшують розуміння програми іншими розробниками та спрощують її модифікацію.

Після завершення написання коду виконується етап тестування програмного забезпечення. Метою тестування є перевірка правильності роботи програми та виявлення можливих помилок.

Під час тестування перевіряється відповідність роботи програми сформованим вимогам. Тестування передбачає виконання програми з різними наборами вхідних даних та аналіз отриманих результатів. Якщо під час тестування виявляються помилки, програмний код виправляється та перевіряється повторно.

Існують різні види тестування програмного забезпечення. Модульне тестування спрямоване на перевірку окремих компонентів програми. Інтеграційне тестування перевіряє взаємодію між різними модулями системи. Системне тестування охоплює перевірку роботи всієї програмної системи в цілому.

Окремим видом є тестування з боку користувача. У цьому випадку програмний продукт перевіряється в умовах, максимально наближених до реальної експлуатації. Такий підхід дає змогу оцінити зручність використання системи та її відповідність потребам користувачів.

Після завершення тестування програмний продукт переходить до етапу експлуатації та супроводу. Супровід програмного забезпечення передбачає підтримку його працездатності, виправлення виявлених помилок та внесення змін відповідно до нових вимог користувачів.

У процесі експлуатації програмної системи можуть виникати нові задачі або змінюватися умови її використання. У таких випадках програмне забезпечення модифікується або розширюється. Таким чином, процес розробки програмного продукту фактично продовжується протягом усього періоду його використання.

Сукупність усіх етапів створення та використання програмної системи утворює її життєвий цикл. Життєвий цикл програмного забезпечення охоплює всі процеси, пов’язані з розробкою, впровадженням, експлуатацією та модернізацією програмного продукту.

Життєвий цикл програмного забезпечення зазвичай поділяється на кілька основних фаз. До них належать аналіз вимог, проєктування, реалізація, тестування, впровадження та супровід.

Фаза аналізу вимог передбачає визначення функцій програмної системи та формування технічного завдання. Фаза проєктування охоплює розробку архітектури системи та алгоритмів її роботи. Фаза реалізації включає написання програмного коду та створення

програмних модулів. Фаза тестування передбачає перевірку правильності роботи програмного продукту. Після цього система впроваджується у середовище експлуатації, де вона використовується користувачами. Завершальною фазою є супровід програмного забезпечення.

Для організації процесу розробки програмного забезпечення використовуються різні методології. Методологія визначає порядок виконання етапів життєвого циклу, способи організації роботи команди розробників та правила управління проєктом.

Однією з класичних моделей життєвого циклу програмного забезпечення є каскадна модель. У цій моделі всі етапи розробки виконуються послідовно. Спочатку формуються вимоги, потім виконується проєктування, після цього здійснюється реалізація програмного коду, тестування та впровадження системи.

Перевагою каскадної моделі є її простота та чітка структура. Кожен етап має визначений результат, що полегшує контроль процесу розробки. Проте така модель має і певні обмеження. Якщо вимоги до системи змінюються під час розробки, внесення змін може бути складним і потребувати значних витрат часу.

Для подолання цих обмежень були розроблені інші методології створення програмного забезпечення. Однією з таких методологій є ітераційна модель. У цьому випадку розробка програмної системи відбувається у вигляді послідовних ітерацій. На кожній ітерації створюється частина функціональності системи, яка поступово розширюється.

Ітераційний підхід дає змогу швидше отримати працездатну версію програмного продукту та поступово вдосконалювати його. Такий підхід також спрощує внесення змін до вимог, оскільки кожна нова ітерація може враховувати нові потреби користувачів.

У сучасній розробці програмного забезпечення широко використовуються гнучкі методології, які орієнтовані на швидку адаптацію до змін вимог. До таких методологій належать підходи, що передбачають короткі цикли розробки, регулярне тестування та постійну взаємодію з користувачами системи.

Гнучкі методології передбачають поділ роботи на невеликі етапи, кожен з яких завершується створенням працездатної частини програмного продукту. Це дає змогу регулярно отримувати результати роботи та оцінювати правильність обраного напрямку розробки.

Вибір методології розробки залежить від багатьох факторів, серед яких складність програмної системи, обсяг проєкту, кількість учасників команди та вимоги замовника. Для великих проєктів часто використовуються комбіновані підходи, які поєднують елементи різних моделей життєвого циклу.

Таким чином, розробка програмного забезпечення є складним процесом, що включає кілька взаємопов’язаних етапів. Формування вимог визначає функціональні можливості майбутньої системи. Проєктування алгоритмів і архітектури забезпечує логічну структуру програмного продукту. Реалізація програмного коду перетворює алгоритми на програму, придатну для виконання комп’ютером. Тестування забезпечує перевірку правильності роботи системи, а супровід підтримує її працездатність у процесі експлуатації. Сукупність цих процесів формує життєвий цикл програмного забезпечення, який визначає організацію роботи під час створення сучасних програмних систем.

ТЕМА 2. ОСНОВИ ПРОГРАМУВАННЯ

2.1 Структура програми мовою C/C++ та основні елементи програмного коду

Мови програмування C та C++ належать до найбільш поширених мов системного та прикладного програмування. Вони широко використовуються для створення операційних систем, прикладних програм, вбудованих систем, а також програмного забезпечення для мікроконтролерів. Вивчення цих мов починається з розуміння структури програми, основних синтаксичних елементів та принципів її компіляції і виконання.

Будь-яка програма мовою C або C++ являє собою текстовий файл, який містить послідовність інструкцій, записаних відповідно до правил мови програмування. Цей текст називають вихідним кодом програми. Вихідний код створюється програмістом у текстовому редакторі або у спеціалізованому середовищі розробки.

Структура програми мовою C/C++ визначає порядок розміщення різних елементів програмного коду. До таких елементів належать директиви препроцесора, оголошення змінних і функцій, визначення функцій та інші програмні конструкції. Хоча конкретна структура програми може відрізнятися залежно від задачі, більшість програм мають подібну загальну організацію.

Зазвичай програма починається з директив препроцесора. Директиви препроцесора використовуються для підключення заголовкових файлів, визначення макросів та виконання інших підготовчих дій перед компіляцією. Наприклад, за допомогою директиви include до програми підключаються бібліотеки стандартних функцій.

Після директив препроцесора можуть розміщуватися глобальні оголошення змінних або функцій. Глобальні змінні доступні для використання у різних частинах програми. Однак у більшості випадків рекомендується обмежувати використання глобальних змінних, оскільки вони ускладнюють структуру програмного коду.

Основною складовою будь-якої програми є функції. Функція являє собою окремий програмний блок, який виконує певну задачу. Використання функцій дає змогу розбивати складну програму на менші частини, що спрощує її розробку, тестування та супровід.

Особливе значення у програмі має функція main. Саме з виконання цієї функції починається робота будь-якої програми мовою C або C++. Операційна система передає керування програмі, викликаючи функцію main, після чого виконання відбувається відповідно до команд, записаних у її тілі.

Функція main має стандартну структуру. Вона містить заголовок функції та її тіло. Заголовок функції визначає тип значення, яке повертає функція, її ім’я та список параметрів. У більшості програм функція main має тип int, що означає повернення цілого значення операційній системі.

Повернене значення функції main використовується для повідомлення операційній системі про результат виконання програми. Зазвичай значення 0 означає успішне завершення роботи програми.

Тіло функції main містить програмні інструкції, які виконуються під час роботи програми. Ці інструкції записуються у фігурних дужках. Усередині функції можуть оголошуватися змінні, виконуватися обчислення, викликатися інші функції та виконуватися різні операції обробки даних.

Для того щоб програма могла бути виконана комп’ютером, її вихідний код повинен пройти процес компіляції. Компіляція — це перетворення тексту програми мовою програмування на машинний код, який може виконуватися процесором комп’ютера.

Процес компіляції складається з кількох етапів. Першим етапом є робота препроцесора. На цьому етапі обробляються директиви препроцесора. Наприклад, директиви підключення заголовкових файлів замінюються відповідним текстом цих файлів. Також виконуються операції заміни макросів та інші підготовчі дії.

Після завершення роботи препроцесора починається власне компіляція програми. На цьому етапі компілятор аналізує текст програми, перевіряє правильність синтаксису та перетворює програму у проміжне представлення. Якщо у програмі містяться синтаксичні помилки, компілятор повідомляє про них програміста.

Після компіляції відбувається етап компонування. Під час компонування окремі частини програми об’єднуються в один виконуваний файл. На цьому етапі також відбувається підключення бібліотечних функцій, які використовуються у програмі.

Результатом роботи компілятора та компонувальника є виконуваний файл програми. Саме цей файл запускається операційною системою під час виконання програми.

Після запуску виконуваного файлу починається виконання програми. Операційна система завантажує програму в оперативну пам’ять і передає керування функції main. Далі виконання відбувається послідовно відповідно до інструкцій програмного коду.

Мова програмування має власний алфавіт — набір символів, які можуть використовуватися під час запису програм. Алфавіт мови C/C++ включає літери латинського алфавіту, цифри, спеціальні символи та символи пробілів.

До літер належать великі та малі літери латинського алфавіту. Важливою особливістю мови C/C++ є розрізнення великих і малих літер. Це означає, що імена змінних, записані різними регістрами, розглядаються як різні ідентифікатори. Наприклад, змінні sum та Sum є різними.

До алфавіту мови також належать цифри від 0 до 9. Цифри використовуються під час запису числових констант, а також можуть входити до складу ідентифікаторів.

Крім літер і цифр, у мові C/C++ використовуються спеціальні символи. До них належать символи арифметичних операцій, дужки, коми, крапки з комою, символи порівняння та інші знаки. Ці символи використовуються для формування операторів, виразів та інших програмних конструкцій.

Окрему роль відіграють символи пробілів. До них належать пробіл, символ табуляції та символ переходу на новий рядок. Ці символи використовуються для розділення елементів програми та покращення читабельності коду. У більшості випадків компілятор не враховує кількість пробілів між елементами програми.

Одним із основних елементів програмного коду є ідентифікатори. Ідентифікатор — це ім’я, яке використовується для позначення змінних, функцій, типів даних та інших програмних об’єктів.

Ідентифікатори повинні відповідати певним правилам. Вони можуть складатися з літер латинського алфавіту, цифр та символу підкреслення. Ідентифікатор повинен починатися з літери або символу підкреслення. Починати ідентифікатор з цифри не дозволяється.

Імена ідентифікаторів повинні бути унікальними у межах відповідної області видимості. Крім того, не дозволяється використовувати зарезервовані слова мови

програмування як ідентифікатори. Зарезервовані слова мають спеціальне призначення і використовуються для запису операторів та інших конструкцій мови.

Для підвищення зрозумілості програмного коду програмісти зазвичай використовують змістовні імена ідентифікаторів. Наприклад, для змінної, що зберігає суму значень, доцільно використовувати ім’я sum або total, а не довільний набір символів.

Ще одним важливим елементом програмного коду є коментарі. Коментарі — це текстові пояснення, які додаються до програми для полегшення її розуміння. Коментарі не впливають на виконання програми, оскільки компілятор ігнорує їх під час обробки вихідного коду.

У мові C/C++ існує два основні види коментарів. Перший вид — це однорядкові коментарі. Вони починаються символами подвійної косої риски. Усе, що розміщується після цих символів до кінця рядка, вважається коментарем.

Другий вид — це багаторядкові коментарі. Вони починаються символами косої риски та зірочки і завершуються символами зірочки та косої риски. Такий тип коментарів може охоплювати кілька рядків тексту.

Коментарі використовуються для пояснення логіки роботи програми, опису призначення змінних та функцій, а також для документування складних фрагментів коду. Наявність зрозумілих коментарів значно полегшує роботу з програмним кодом, особливо у великих програмних проєктах.

Під час написання програм мовою C/C++ важливо дотримуватися певних правил структурування коду. Зокрема, програмний код повинен бути логічно впорядкований, мати зрозумілу структуру та бути зручним для читання. Для цього використовуються відступи, поділ коду на логічні блоки та зрозумілі імена змінних і функцій.

Таким чином, структура програми мовою C/C++ включає директиви препроцесора, оголошення змінних і функцій, а також визначення функцій, серед яких центральну роль відіграє функція main. Перед виконанням програма проходить кілька етапів обробки, включаючи роботу препроцесора, компіляцію та компонування. Основними елементами програмного коду є символи алфавіту мови, ідентифікатори та коментарі. Розуміння цих елементів є необхідною основою для подальшого вивчення програмування мовами C та C++ і створення програмних систем різної складності.

2.2 Типи даних, змінні та операції у програмах мовою C/C++

Будь-яка програма виконує обробку певних даних. Дані можуть мати різну природу: числові значення, символи, текстові рядки, логічні значення або складні структури. Для правильного представлення та обробки інформації у мовах програмування використовуються типи даних. Тип даних визначає спосіб зберігання значення в пам’яті комп’ютера, допустимі операції над цими значеннями та діапазон можливих значень.

У мовах програмування C та C++ типи даних відіграють важливу роль, оскільки ці мови належать до мов із суворою типізацією. Це означає, що кожна змінна повинна мати визначений тип, який задається під час її оголошення. Тип даних визначає обсяг пам’яті, необхідний для зберігання значення, а також операції, які можуть виконуватися над цією змінною.

До базових типів даних мов C і C++ належать цілі числа, числа з плаваючою комою, символи та логічні значення. Цілі типи використовуються для зберігання цілих чисел без

дробової частини. До них належать типи short, int, long та long long. Ці типи відрізняються між собою діапазоном можливих значень і кількістю байтів пам'яті, необхідних для їх зберігання.

Тип int є одним із найбільш часто використовуваних типів у програмуванні. Він використовується для представлення цілих чисел у широкому діапазоні значень. У багатьох випадках саме цей тип використовується за замовчуванням для арифметичних обчислень.

Типи з плаваючою комою використовуються для представлення дійсних чисел. До них належать типи float, double та long double. Вони дають змогу зберігати числа, що мають дробову частину. Такі типи використовуються під час виконання наукових обчислень, обробки вимірювальних даних та інших задач, де потрібна підвищена точність представлення чисел.

Тип char використовується для представлення символів. Кожне значення цього типу відповідає певному символу таблиці кодування. Фактично символи зберігаються у пам'яті як числові коди.

У мові C++ також використовується логічний тип bool, який може набувати лише двох значень: істина або хиба. Цей тип використовується під час виконання логічних операцій та перевірки умов у програмі.

Окрім базових типів, мови C і C++ підтримують модифікатори типів. Наприклад, модифікатори signed та unsigned визначають, чи може змінна набувати від'ємних значень. Тип unsigned використовується для представлення лише невід'ємних чисел, що дає змогу розширити верхню межу діапазону значень.

Для зберігання даних у програмі використовуються змінні. Змінна — це іменована область пам'яті, у якій зберігається певне значення. Кожна змінна має ім'я, тип та значення.

Перед використанням змінної у програмі її необхідно оголосити. Оголошення змінної передбачає визначення її типу та імені. Під час оголошення компілятор виділяє відповідний обсяг пам'яті для зберігання значення цієї змінної.

Ім'я змінної повинно відповідати правилам формування ідентифікаторів. Воно може складатися з літер латинського алфавіту, цифр та символу підкреслення. Ім'я змінної повинно починатися з літери або символу підкреслення.

У програмуванні важливо використовувати зрозумілі імена змінних, які відображають їх призначення. Наприклад, для змінної, що зберігає кількість елементів, доцільно використовувати ім'я count, а для змінної, що зберігає суму значень, — sum.

Змінні можуть отримувати значення під час оголошення або у процесі виконання програми. Процес присвоювання значення змінній називається ініціалізацією. Ініціалізація дає змогу одразу задати початкове значення змінної.

Окрім змінних, у програмуванні використовуються константи. Константа — це значення, яке не змінюється під час виконання програми. Константи використовуються для представлення фіксованих значень, таких як математичні коефіцієнти, параметри системи або інші постійні величини.

У мовах C і C++ константи можуть бути представлені у вигляді літералів або іменованих констант. Літерали — це значення, які безпосередньо записуються у програмному коді. Наприклад, числові значення або символи.

Іменовані константи створюються за допомогою спеціальних механізмів мови програмування. У мові C++ для цього часто використовується ключове слово const. Такі константи мають власні імена, що підвищує зрозумілість програмного коду.

Під час написання програм важливе значення має область видимості змінних. Область видимості визначає частину програми, у межах якої змінна може використовуватися.

Залежно від місця оголошення змінної розрізняють глобальну та локальну області видимості. Глобальні змінні оголошуються поза межами функцій і доступні для використання у всій програмі. Локальні змінні оголошуються всередині функції або блоку коду і доступні лише в межах цього блоку.

Локальні змінні створюються під час входу у відповідний блок програми та знищуються після завершення його виконання. Такий підхід дає змогу обмежити доступ до змінних і уникнути небажаних змін їх значень у різних частинах програми.

Використання локальних змінних є важливим принципом структурованого програмування. Це дає змогу розділяти програму на незалежні частини та зменшувати ймовірність помилок.

Для обробки числових значень у програмі використовуються арифметичні операції. До основних арифметичних операцій належать додавання, віднімання, множення та ділення. Ці операції використовуються для виконання математичних обчислень над числовими значеннями.

Окрім базових арифметичних операцій, у мовах C і C++ використовуються операції отримання остачі від ділення. Така операція застосовується лише до цілих чисел і використовується у задачах, пов’язаних із перевіркою кратності чисел або виконанням циклічних обчислень.

Важливим елементом програмування є оператор присвоювання. Оператор присвоювання використовується для запису значення у змінну. Під час виконання цього оператора значення виразу, що знаходиться праворуч від оператора, обчислюється та записується у змінну, яка знаходиться ліворуч.

Оператор присвоювання є одним із найчастіше використовуваних операторів у програмуванні. Він використовується для зміни значень змінних, виконання обчислень та організації логіки програми.

У мовах C і C++ також використовуються складені оператори присвоювання. Вони поєднують арифметичну операцію та присвоювання в одну інструкцію. Такі оператори скорочують запис програмного коду та роблять його більш компактним.

Під час виконання арифметичних виразів важливе значення має порядок виконання операцій. У мовах програмування, як і в математиці, існує певний пріоритет операцій. Операції з вищим пріоритетом виконуються раніше за операції з нижчим пріоритетом.

Найвищий пріоритет мають операції, що виконуються у дужках. Використання дужок дає змогу змінювати порядок виконання операцій і робити вирази більш зрозумілими.

Після операцій у дужках виконуються операції множення та ділення. Далі виконуються операції додавання та віднімання. Якщо у виразі є кілька операцій з однаковим пріоритетом, вони виконуються у порядку їх появи зліва направо.

Розуміння пріоритету операцій є важливим для правильного запису програмних виразів. Неправильний порядок операцій може призвести до отримання неправильних результатів обчислень.

Ще одним важливим аспектом роботи з даними є перетворення типів. Перетворення типів — це зміна типу значення під час виконання програми. Таке перетворення може відбуватися автоматично або виконуватися програмістом явно.

Автоматичне перетворення типів виконується компілятором під час обчислення виразів, у яких використовуються значення різних типів. У таких випадках компілятор приводить значення до спільного типу, щоб виконати обчислення.

Наприклад, якщо у виразі використовується ціле число і число з плаваючою комою, ціле число автоматично перетворюється на дійсне. Це дає змогу виконати обчислення з використанням дробової частини.

Явне перетворення типів виконується програмістом за допомогою спеціальних конструкцій мови програмування. Такий підхід використовується у випадках, коли необхідно явно змінити тип значення для виконання певної операції.

Перетворення типів використовується у багатьох програмних задачах. Наприклад, під час обчислення середнього значення чисел може виникнути потреба перетворити цілі значення на дійсні, щоб уникнути втрати дробової частини результату.

Водночас використання перетворень типів потребує обережності. У деяких випадках зміна типу може призвести до втрати точності або некоректного представлення значення.

Таким чином, типи даних є основою представлення інформації у програмах мовою C/C++. Змінні використовуються для зберігання значень, які можуть змінюватися під час виконання програми, а константи застосовуються для представлення незмінних величин. Область видимості визначає доступність змінних у різних частинах програми. Арифметичні операції та оператори присвоювання використовуються для виконання обчислень і зміни значень змінних. Важливе значення має також пріоритет операцій, який визначає порядок виконання виразів. Перетворення типів дає змогу виконувати обчислення між значеннями різних типів. Розуміння цих понять є необхідною основою для подальшого вивчення програмування мовами C та C++ і створення програмних систем різної складності.

2.3 Логічні вирази, введення та виведення даних і реалізація лінійних алгоритмів у мовах C/C++

Під час розробки програм важливу роль відіграє робота з умовами, порівняннями та логічними перевірками. Програма повинна вміти аналізувати дані, перевіряти їх значення та приймати рішення залежно від результатів цих перевірок. Для цього у мовах програмування використовуються операції відношення та логічні операції. На основі таких операцій формуються логічні вирази, які широко застосовуються у програмуванні.

Логічні операції та операції відношення дозволяють порівнювати значення змінних і визначати, чи виконується певна умова. Результатом виконання таких операцій є логічне значення. У мовах C і C++ логічне значення може інтерпретуватися як істина або хиба.

Операції відношення використовуються для порівняння двох значень. До основних операцій відношення належать перевірка рівності, перевірка нерівності, перевірка більшого або меншого значення, а також перевірка відношення більше або дорівнює та менше або дорівнює.

Операція рівності використовується для перевірки того, чи мають два значення однакове числове або символьне представлення. Якщо значення однакові, результат операції є істинним. Якщо значення відрізняються, результат буде хибним.

Операція нерівності виконує протилежну перевірку. Вона використовується для визначення того, чи відрізняються два значення. Якщо значення не рівні, результат операції є істинним.

Операції більше та менше використовуються для порівняння числових значень. Вони дозволяють визначити, яке з двох значень є більшим або меншим. Такі операції часто використовуються під час аналізу даних, пошуку максимального або мінімального значення та перевірки виконання певних умов.

Операції більше або дорівнює та менше або дорівнює дозволяють перевіряти одночасно два можливі варіанти: рівність значень або виконання відповідного відношення.

Операції відношення використовуються для формування умов у програмі. Результат таких операцій зазвичай використовується у логічних виразах або у конструкціях керування виконанням програми.

Окрім операцій відношення, у програмуванні широко використовуються логічні операції. Логічні операції дозволяють комбінувати кілька умов і формувати складні логічні вирази.

До основних логічних операцій належать логічне заперечення, логічне множення та логічне додавання. Логічне заперечення змінює значення умови на протилежне. Якщо умова є істинною, результат операції заперечення буде хибним, і навпаки.

Логічне множення використовується для об’єднання двох умов таким чином, що результат буде істинним лише у випадку, коли обидві умови виконуються одночасно. Якщо хоча б одна з умов є хибною, результат логічного множення також буде хибним.

Логічне додавання використовується для перевірки того, чи виконується хоча б одна з кількох умов. Якщо хоча б одна умова є істинною, результат операції буде істинним.

Логічні операції дають змогу створювати складні умови, які використовуються у програмі для аналізу даних. Наприклад, під час перевірки коректності введених значень можна перевірити, чи знаходиться число у певному діапазоні значень.

Поєднання операцій відношення та логічних операцій утворює логічні вирази. Логічний вираз — це вираз, результатом якого є логічне значення. Такі вирази широко використовуються у програмних конструкціях, що керують виконанням програми.

Під час побудови логічних виразів важливо враховувати порядок виконання операцій. Спочатку виконуються операції відношення, після чого застосовуються логічні операції. Використання дужок дозволяє змінювати порядок виконання операцій і робити логічні вирази більш зрозумілими.

Логічні вирази широко застосовуються у задачах, пов’язаних із перевіркою умов, фільтрацією даних та аналізом значень змінних. Вони дозволяють формувати складні правила обробки інформації.

Ще одним важливим аспектом програмування є введення та виведення даних. Більшість програм повинні взаємодіяти з користувачем або з іншими програмами. Для цього необхідно організувати введення даних у програму та виведення результатів її роботи.

У мовах C і C++ введення та виведення даних можуть виконуватися різними способами. Один із найбільш поширених способів передбачає використання стандартних бібліотечних функцій.

Для введення даних у програмі використовується зчитування інформації з клавіатури або з інших джерел. Під час введення дані записуються у змінні, після чого вони можуть використовуватися у програмі для виконання обчислень або інших операцій.

Виведення даних передбачає відображення результатів роботи програми на екрані або запис їх у файл. Виведення результатів є важливою частиною програмного забезпечення, оскільки саме за допомогою цього механізму користувач отримує інформацію про результати обчислень.

У мові С для введення та виведення даних широко використовуються функції стандартної бібліотеки введення-виведення. Ці функції дозволяють зчитувати числові значення, символи та текстові дані.

У мові С++ часто використовуються потоки введення та виведення, які забезпечують більш зручний і безпечний механізм обміну даними. Потоки дозволяють виконувати введення та виведення різних типів даних без необхідності використання складних форматних рядків.

Під час введення даних важливо перевіряти їх коректність. Неправильні або некоректні дані можуть призвести до помилок у роботі програми або до отримання неправильних результатів обчислень.

Окрім самого введення і виведення даних, важливим є питання форматування даних. Форматування передбачає керування способом відображення значень під час їх виведення.

Форматування дозволяє визначити кількість знаків після коми для дійсних чисел, ширину поля виведення, вирівнювання тексту та інші параметри відображення даних. Це особливо важливо під час створення програм, що формують звіти або відображають великі обсяги інформації.

Наприклад, під час виведення числових значень може виникнути потреба обмежити кількість знаків після коми. Це дозволяє зробити результати обчислень більш зрозумілими для користувача.

Форматування також використовується під час виведення таблиць даних. У таких випадках важливо забезпечити однакову ширину колонок та правильне вирівнювання значень.

Після того як програма отримує вхідні дані, вона виконує певні обчислення або інші операції. Найпростішим видом алгоритмів, що реалізуються у програмуванні, є лінійні алгоритми.

Лінійний алгоритм — це алгоритм, у якому всі операції виконуються послідовно одна за одною без розгалужень і повторень. У такому алгоритмі кожна наступна операція виконується після завершення попередньої.

Лінійні алгоритми використовуються у задачах, де необхідно виконати послідовність обчислень без перевірки умов або повторення дій. Наприклад, обчислення значення математичного виразу або перетворення одиниць вимірювання може бути реалізовано у вигляді лінійного алгоритму.

Реалізація лінійного алгоритму у програмі зазвичай складається з кількох етапів. Спочатку оголошуються змінні, у яких будуть зберігатися вхідні дані та результати обчислень. Далі виконується введення початкових значень. Після цього виконуються необхідні обчислення. Завершальним етапом є виведення отриманих результатів.

Лінійні алгоритми є основою для вивчення програмування, оскільки вони дозволяють зосередитися на роботі з даними та виконанні обчислень. У подальшому такі алгоритми можуть ускладнюватися за рахунок додавання умовних операторів та циклів.

Під час реалізації лінійних алгоритмів важливо правильно організовувати послідовність виконання операцій. Порушення порядку обчислень може призвести до неправильних результатів.

Для підвищення зрозумілості програмного коду програмісти часто використовують коментарі та логічну структуру коду. Це дозволяє швидше зрозуміти призначення окремих частин програми та спрощує її подальше редагування.

Таким чином, логічні операції та операції відношення є важливими інструментами аналізу даних у програмі. Вони дозволяють порівнювати значення змінних і формувати логічні вирази, які використовуються для перевірки умов. Введення та виведення даних забезпечують взаємодію програми з користувачем або іншими системами. Форматування даних дозволяє керувати способом відображення інформації під час виведення результатів. Лінійні алгоритми є найпростішим видом алгоритмів і використовуються для реалізації послідовних обчислень без розгалужень та повторень. Розуміння цих принципів є важливим етапом формування практичних навичок програмування мовами C і C++.

ТЕМА 3. РОЗГАЛУЖЕНІ ТА ЦИКЛІЧНІ АЛГОРИТМИ

3.1 Реалізація розгалужених алгоритмів

Під час створення програм часто виникають задачі, у яких подальші дії залежать від певних умов. Наприклад, програма може виконувати різні обчислення залежно від значення змінної, перевіряти правильність введених даних або визначати категорію значення за заданими критеріями. Для реалізації таких задач у програмуванні використовуються розгалужені алгоритми. Розгалужений алгоритм передбачає вибір одного з кількох можливих шляхів виконання залежно від результату перевірки певної умови. У мовах програмування C і C++ для реалізації розгалужень використовуються умовні оператори, основними з яких є оператор if та оператор switch.

Розгалуження є однією з базових алгоритмічних структур поряд із послідовністю та повторенням. Використання розгалужень дозволяє програмі аналізувати значення змінних і виконувати різні дії залежно від отриманих результатів. Завдяки цьому програма може адаптувати свою поведінку до різних ситуацій, що є необхідною умовою для створення складних програмних систем.

Найбільш поширеним засобом реалізації розгалужень у мовах C і C++ є умовний оператор if. Цей оператор використовується для перевірки певної логічної умови. Якщо умова виконується, програма виконує одну групу інструкцій. Якщо умова не виконується, виконання програми продовжується іншим чином. Таким чином, оператор if дозволяє реалізувати альтернативні варіанти виконання програми.

Умовний оператор if має дві основні форми: скорочену та повну. Скорочена форма використовується у випадках, коли необхідно виконати певну дію лише за умови виконання логічного виразу. У цьому випадку оператор містить лише перевірку умови та один блок команд, який виконується у разі істинності цієї умови. Якщо умова не виконується, жодних додаткових дій не відбувається, і програма переходить до виконання наступних інструкцій.

Скорочена форма умовного оператора використовується у тих випадках, коли не потрібно виконувати альтернативні дії у разі невиконання умови. Наприклад, програма може перевіряти, чи перевищує значення змінної певну межу, і виконувати обчислення лише у разі виконання цієї умови. Такий підхід дозволяє спростити структуру програмного коду та зробити його більш зрозумілим.

Повна форма умовного оператора if передбачає використання двох альтернативних гілок виконання. У цьому випадку, якщо умова виконується, виконується один блок інструкцій, а якщо умова не виконується — інший блок. Така структура дозволяє програмі виконувати різні дії залежно від результату перевірки умови.

Повна форма умовного оператора широко використовується у програмуванні, оскільки більшість задач потребує обробки як позитивного, так і альтернативного варіанту виконання. Наприклад, програма може перевіряти правильність введених даних і виконувати різні дії залежно від того, чи відповідають ці дані заданим вимогам.

Важливим аспектом використання умовного оператора є правильне формування логічної умови. Умова зазвичай містить операції відношення або логічні операції. Результатом такої умови є логічне значення, яке визначає подальший хід виконання програми.

У складніших задачах може виникнути необхідність перевіряти кілька умов одночасно. Для цього використовуються логічні операції, які дозволяють поєднувати кілька умов у єдиний логічний вираз. Такий підхід дозволяє реалізувати складні правила обробки даних.

Ще одним важливим засобом реалізації розгалужень є вкладені умовні оператори. Вкладене розгалуження виникає у випадку, коли один оператор if розміщується всередині іншого. Така структура використовується у тих випадках, коли необхідно перевірити кілька умов послідовно.

Вкладені розгалуження дозволяють створювати складні алгоритми прийняття рішень. Наприклад, програма може спочатку перевіряти, чи належить значення певному діапазону, а потім виконувати додаткові перевірки для уточнення результату. Така структура дозволяє реалізувати багаторівневу логіку аналізу даних.

Водночас використання вкладених розгалужень потребує певної обережності. Надмірна кількість вкладених умов може ускладнювати структуру програми та знижувати її зрозумілість. Тому під час розробки програм рекомендується прагнути до логічно чіткої структури умовних операторів.

Для підвищення зрозумілості програмного коду часто використовуються відступи та блоки інструкцій, обмежені фігурними дужками. Це дозволяє чітко визначити межі виконання кожної гілки умовного оператора і полегшує читання програми.

Окрім оператора if, у мовах C і C++ використовується оператор switch, який також призначений для реалізації розгалужених алгоритмів. Оператор switch застосовується у випадках, коли необхідно виконати одну з кількох можливих дій залежно від значення певної змінної або виразу.

Оператор switch зручний у тих ситуаціях, коли потрібно порівняти значення змінної з кількома можливими варіантами. Кожен можливий варіант значення обробляється окремою гілкою виконання. Такий підхід дозволяє значно спростити структуру програмного коду у випадках, коли потрібно перевіряти багато альтернатив.

Структура оператора switch складається з виразу, значення якого перевіряється, та набору альтернативних варіантів виконання. Кожен варіант відповідає певному значенню перевірюваного виразу. Якщо значення виразу збігається з одним із заданих варіантів, виконується відповідний блок інструкцій.

Окрім варіантів, оператор switch може містити спеціальний блок, який виконується у випадку, якщо жоден із варіантів не відповідає значенню виразу. Такий блок використовується для обробки всіх інших можливих ситуацій.

Використання оператора switch є особливо ефективним у задачах, де необхідно обробляти різні значення змінної, наприклад під час вибору пункту меню або визначення типу операції. У таких випадках оператор switch дозволяє уникнути великої кількості вкладених умовних операторів if.

Під час використання оператора switch важливо враховувати особливості його роботи. Зокрема, після виконання певної гілки програма може продовжити виконання наступних інструкцій, якщо не використовується спеціальна команда завершення гілки. Тому під час розробки програм необхідно уважно контролювати логіку виконання операторів у межах конструкції switch.

Аналіз логіки розгалужених алгоритмів є важливою частиною процесу програмування. Перед реалізацією алгоритму у вигляді програмного коду необхідно чітко визначити всі можливі варіанти розвитку подій і відповідні дії, які повинна виконувати програма.

Під час аналізу розгалужених алгоритмів програміст визначає умови, які впливають на вибір подальших дій. Для кожної умови необхідно визначити можливі результати та відповідні дії програми. Такий підхід дозволяє створити логічно завершену структуру алгоритму.

Для аналізу розгалужених алгоритмів часто використовуються блок-схеми або псевдокод. Ці інструменти дозволяють наочно відобразити структуру алгоритму та взаємозв'язок між різними гілками виконання. Використання таких засобів значно спрощує процес проєктування програмного забезпечення.

Під час аналізу логіки розгалужених алгоритмів також важливо враховувати можливість виникнення крайніх випадків. Наприклад, необхідно передбачити ситуації, коли значення змінної виходить за межі очікуваного діапазону або коли введені дані є некоректними.

Ще одним важливим аспектом є перевірка повноти умов. Усі можливі варіанти значень повинні бути оброблені програмою. Якщо певний варіант не врахований у логіці алгоритму, це може призвести до помилок або некоректної роботи програми.

Під час реалізації розгалужених алгоритмів необхідно також звертати увагу на зрозумілість програмного коду. Чітка структура умовних операторів, використання змістовних ідентифікаторів та логічне групування інструкцій дозволяють значно полегшити читання та супровід програми.

Таким чином, умовні оператори є важливим засобом реалізації розгалужених алгоритмів у програмуванні. Оператор if дозволяє перевіряти умови та виконувати різні дії залежно від результату перевірки. Повна та скорочена форми цього оператора забезпечують гнучкість під час реалізації різних варіантів логіки програми. Вкладені розгалуження дозволяють реалізувати складні алгоритми прийняття рішень. Оператор switch використовується для вибору однієї з кількох альтернатив залежно від значення змінної. Аналіз логіки розгалужених алгоритмів є важливим етапом розробки програмного забезпечення, оскільки він дозволяє забезпечити правильну та повну обробку всіх можливих варіантів виконання програми.

3.2 Оператори циклу у мовах C/C++ та організація повторюваних обчислень

У програмуванні дуже часто виникають задачі, у яких певні дії необхідно виконувати багаторазово. Наприклад, може виникнути потреба обробити послідовність чисел, виконати багаторазові обчислення або перевіряти умову до тих пір, поки вона не зміниться. Реалізація таких задач у вигляді послідовного запису однакових інструкцій є неефективною та ускладнює програмний код. Для вирішення цієї проблеми використовуються циклічні алгоритми. Циклічний алгоритм передбачає повторення певної групи інструкцій до

виконання заданої умови завершення. У мовах програмування C і C++ для організації повторень використовуються оператори циклів, основними з яких є оператори for, while та do while.

Цикл є однією з базових алгоритмічних структур поряд із послідовністю та розгалуженням. Використання циклів дозволяє програмі виконувати однакові або подібні операції над різними даними або повторювати обчислення до отримання потрібного результату. Завдяки циклам програмний код стає значно компактнішим, зрозумілішим і більш ефективним.

Оператор циклу for є одним із найпоширеніших засобів організації повторень у мовах C і C++. Цей оператор зазвичай використовується у випадках, коли кількість повторень відома заздалегідь або може бути визначена перед початком виконання циклу. Структура оператора for дозволяє об'єднати три важливі елементи керування циклом: початкову ініціалізацію змінної, перевірку умови виконання та зміну значення змінної керування.

На початку роботи циклу for виконується ініціалізація змінної, яка використовується для керування кількістю повторень. Після цього перевіряється умова виконання циклу. Якщо умова є істинною, виконується тіло циклу. Після завершення виконання тіла циклу змінюється значення змінної керування, і процес повторюється. Коли умова стає хибною, виконання циклу припиняється, і програма переходить до наступної інструкції.

Оператор for зручний тим, що всі основні елементи керування циклом зосереджені в одному місці. Це полегшує читання та аналіз програмного коду. Найчастіше оператор for використовується для організації циклів із лічильником, коли певний блок інструкцій необхідно виконати визначену кількість разів.

У багатьох задачах оператор for використовується для обробки елементів масиву або для виконання послідовних обчислень. Наприклад, за допомогою такого циклу можна обчислювати суму певної кількості чисел або формувати послідовність значень.

Ще одним важливим оператором циклу є оператор while. На відміну від оператора for, оператор while використовується у випадках, коли кількість повторень заздалегідь невідома. У цьому випадку виконання циклу продовжується доти, доки виконується певна логічна умова.

Оператор while належить до циклів з передумовою. Це означає, що перевірка умови виконується перед кожним виконанням тіла циклу. Якщо умова є істинною, виконується тіло циклу, після чого перевірка повторюється. Якщо ж умова з самого початку є хибною, тіло циклу не виконується жодного разу.

Цикли з передумовою використовуються у тих випадках, коли необхідно перевірити можливість виконання циклу ще до початку повторень. Наприклад, програма може зчитувати дані до тих пір, поки користувач не введе певне спеціальне значення.

Оператор while широко використовується у задачах, пов'язаних із перевіркою умов завершення процесу. Він також застосовується у випадках, коли завершення циклу залежить від певних подій або результатів обчислень.

Третім поширеним оператором циклу у мовах C і C++ є оператор do while. Цей оператор використовується для організації циклів із післяумовою. У таких циклах перевірка умови виконується після виконання тіла циклу.

Особливістю циклу do while є те, що тіло циклу виконується щонайменше один раз незалежно від значення умови. Після виконання інструкцій у тілі циклу перевіряється умова. Якщо умова є істинною, виконання циклу повторюється. Якщо ж умова є хибною, цикл завершується.

Цикли з післяумовою використовуються у тих випадках, коли певні дії необхідно виконати хоча б один раз. Наприклад, такий цикл може використовуватися для організації меню програми, яке повторюється до тих пір, поки користувач не обере команду завершення роботи.

Порівняння циклів з передумовою та післяумовою є важливим для правильного вибору конструкції циклу під час розробки програм. Основна відмінність між цими типами циклів полягає у моменті перевірки умови.

У циклах з передумовою умова перевіряється перед виконанням тіла циклу. Якщо умова є хибною, цикл не виконується жодного разу. Такий підхід дозволяє уникнути виконання непотрібних дій, якщо умова не виконується з самого початку.

У циклах з післяумовою перевірка умови відбувається після виконання тіла циклу. Це означає, що тіло циклу виконується принаймні один раз. Такий підхід зручний у тих випадках, коли певна дія повинна бути виконана незалежно від початкового значення умови.

Вибір між циклами з передумовою та післяумовою залежить від логіки задачі. Якщо необхідно перевірити умову перед початком повторень, доцільно використовувати цикл while. Якщо ж певні дії повинні бути виконані хоча б один раз, більш доцільним є використання циклу do while.

Організація повторень у програмі потребує уважного проєктування логіки алгоритму. Під час розробки циклічних алгоритмів необхідно чітко визначити умову завершення циклу. Якщо така умова сформульована неправильно або відсутня, програма може потрапити у нескінченний цикл.

Нескінченний цикл виникає у випадку, коли умова завершення ніколи не стає хибною. У результаті програма продовжує виконувати повторення без завершення. Такі ситуації зазвичай виникають через помилки у програмному коді або неправильну зміну значення змінної керування циклом.

Щоб уникнути нескінченних циклів, необхідно забезпечити правильну зміну значень змінних, які використовуються у перевірці умов. Також важливо уважно перевіряти логіку умов завершення циклу.

Ще одним важливим аспектом організації повторень є використання вкладених циклів. Вкладений цикл виникає у випадку, коли один цикл розміщується всередині іншого. Така структура використовується у задачах, де необхідно виконувати повторення у кількох вимірах.

Наприклад, вкладені цикли часто використовуються під час обробки двовимірних масивів або формування таблиць значень. У цьому випадку зовнішній цикл керує переходом між рядками таблиці, а внутрішній цикл — переходом між елементами у межах одного рядка.

Використання вкладених циклів дозволяє реалізувати складні алгоритми обробки даних. Однак надмірна кількість вкладених циклів може ускладнювати структуру програми. Тому під час розробки програм необхідно прагнути до оптимальної структури алгоритму.

Під час роботи з циклами важливо також правильно організовувати програмний код. Використання відступів, логічного групування інструкцій та зрозумілих імен змінних значно полегшує читання програми.

У багатьох випадках цикл поєднується з іншими алгоритмічними конструкціями, такими як умовні оператори. Це дозволяє реалізувати складні алгоритми, у яких повторення поєднується з аналізом умов і прийняттям рішень.

Цикли широко використовуються у різних задачах програмування. Вони застосовуються під час обробки великих масивів даних, виконання чисельних обчислень, аналізу послідовностей значень та реалізації алгоритмів пошуку і сортування.

Таким чином, оператори циклу є важливим інструментом організації повторюваних обчислень у програмуванні. Оператор for використовується переважно у випадках, коли кількість повторень відома заздалегідь. Оператор while застосовується для циклів з передумовою, у яких перевірка умови виконується перед кожним повторенням. Оператор do while використовується для циклів з післяумовою, у яких тіло циклу виконується щонайменше один раз. Порівняння циклів з передумовою та післяумовою дозволяє правильно обирати конструкцію циклу залежно від логіки задачі. Організація повторень є важливою складовою алгоритмізації та програмування, оскільки вона дозволяє ефективно реалізувати обробку великих обсягів даних і складні обчислювальні процеси.

3.3 Вкладені цикли та обробка послідовностей даних у мовах C/C++

Циклічні конструкції є важливою складовою алгоритмізації та програмування. За допомогою циклів у програмі реалізується багаторазове виконання певних дій, що дозволяє ефективно обробляти великі обсяги даних, виконувати повторювані обчислення та організовувати складні алгоритми. У багатьох задачах виникає потреба виконувати повторення не лише на одному рівні, а й організовувати вкладені повторення. У таких випадках використовуються вкладені цикли. Вкладені цикли, обчислення сум і добутків, табулювання функцій та обробка послідовностей даних є типовими задачами, що широко використовуються під час навчання програмуванню та у практичній діяльності програмістів.

Вкладений цикл виникає у випадку, коли один цикл розміщується всередині іншого. У такій структурі зовнішній цикл виконує керування загальною кількістю повторень, тоді як внутрішній цикл виконує повторення у межах кожної ітерації зовнішнього циклу. Іншими словами, для кожного повторення зовнішнього циклу внутрішній цикл виконується повністю.

Вкладені цикли дозволяють реалізовувати алгоритми, у яких обробка даних відбувається у кількох вимірах. Наприклад, така структура часто використовується під час роботи з таблицями значень, двовимірними масивами, матрицями або іншими структурами даних, що мають рядки і стовпці. У таких задачах зовнішній цикл зазвичай відповідає за перехід між рядками, а внутрішній цикл — за обробку елементів у межах одного рядка.

Під час використання вкладених циклів важливо правильно організувати логіку алгоритму. Необхідно чітко визначити, яка змінна керує зовнішнім циклом, а яка — внутрішнім. Також потрібно забезпечити коректну зміну значень цих змінних, щоб уникнути нескінченних циклів або помилок у роботі програми.

Ще одним важливим аспектом використання вкладених циклів є їх вплив на ефективність програмного коду. Оскільки внутрішній цикл виконується для кожної ітерації зовнішнього циклу, загальна кількість виконаних операцій може значно зростати. Тому під час розробки алгоритмів необхідно прагнути до оптимальної структури вкладених повторень.

Однією з найпоширеніших задач, що реалізуються за допомогою циклів, є обчислення сум. Обчислення суми послідовності значень є типовою задачею програмування. Наприклад,

може виникнути потреба обчислити суму перших n чисел, суму елементів масиву або суму значень певної функції на заданому інтервалі.

Для реалізації таких задач зазвичай використовується змінна-накопичувач. Ця змінна зберігає поточне значення суми, яке поступово збільшується під час виконання циклу. На початку роботи програми змінна-накопичувач ініціалізується нульовим значенням. Під час кожної ітерації циклу до неї додається нове значення.

Такий підхід дозволяє поступово накопичувати результат обчислень. Після завершення виконання циклу змінна містить остаточний результат — суму всіх значень, що були оброблені під час роботи програми.

Аналогічним чином організовується обчислення добутків. У цьому випадку використовується змінна-накопичувач, яка зберігає поточний добуток значень. На відміну від обчислення суми, початкове значення такої змінної дорівнює одиниці. Під час кожної ітерації циклу поточне значення множиться на новий елемент послідовності.

Обчислення добутків часто використовується у задачах, пов’язаних із математичними обчисленнями. Наприклад, за допомогою такого підходу можна обчислювати факторіал числа або значення складних математичних виразів.

Обчислення сум і добутків широко застосовується у різних галузях програмування, включаючи статистичну обробку даних, аналіз числових послідовностей та реалізацію математичних алгоритмів. Використання циклів у таких задачах дозволяє реалізувати універсальні алгоритми, які працюють з довільною кількістю елементів.

Ще однією важливою задачею програмування є табулювання функцій. Табулювання функції означає обчислення значень функції для набору значень аргументу. Результати таких обчислень зазвичай подаються у вигляді таблиці, де кожному значенню аргументу відповідає певне значення функції.

Табулювання функцій широко використовується у наукових і технічних обчисленнях. Наприклад, під час аналізу математичних моделей може виникнути потреба обчислити значення функції на певному інтервалі з заданим кроком зміни аргументу.

Реалізація табулювання функції у програмі зазвичай виконується за допомогою циклу. На початку задається початкове значення аргументу, кінцеве значення та крок зміни. Під час кожної ітерації циклу обчислюється значення функції для поточного аргументу. Після цього аргумент змінюється на величину кроку, і процес повторюється.

Результати табулювання можуть виводитися у вигляді таблиці значень. У такій таблиці кожний рядок містить значення аргументу та відповідне значення функції. Для покращення читабельності результатів часто використовується форматування даних.

Табулювання функцій дозволяє аналізувати поведінку математичних залежностей і отримувати наближені результати обчислень. Такий підхід широко використовується у чисельних методах, моделюванні фізичних процесів та аналізі експериментальних даних.

Ще однією важливою сферою застосування циклів є обробка послідовностей даних. У програмуванні послідовність даних може бути представлена у вигляді набору значень, які обробляються по одному. Така послідовність може зчитуватися з клавіатури, зберігатися у масиві або надходити з файлу.

Під час обробки послідовностей даних часто виникає потреба виконувати різні операції над кожним елементом. Наприклад, програма може визначати максимальне або мінімальне значення, підраховувати кількість елементів, що задовольняють певну умову, або обчислювати середнє значення послідовності.

Для реалізації таких задач використовується цикл, який послідовно обробляє всі елементи послідовності. Під час кожної ітерації циклу виконується аналіз поточного значення та виконуються необхідні обчислення.

Наприклад, під час пошуку максимального значення послідовності програма порівнює кожен елемент із поточним максимальним значенням. Якщо нове значення більше за поточне, воно стає новим максимумом.

Аналогічним чином можна реалізувати пошук мінімального значення, підрахунок кількості елементів або інші операції обробки даних.

Під час обробки послідовностей даних важливо правильно організувати введення інформації. У деяких задачах кількість елементів відома заздалегідь, і цикл виконується певну кількість разів. У інших випадках кількість елементів може бути невідомою, і завершення обробки визначається спеціальною умовою або певним значенням.

Обробка послідовностей даних є важливою складовою багатьох програмних систем. Вона використовується у задачах статистичного аналізу, обробки сигналів, аналізу результатів вимірювань та у багатьох інших сферах.

Під час реалізації таких алгоритмів важливо дотримуватися чіткої структури програмного коду. Використання зрозумілих імен змінних, логічного групування інструкцій та коментарів значно полегшує розуміння програми та її подальшу модифікацію.

Таким чином, вкладені цикли є важливим інструментом реалізації складних алгоритмів повторення. Вони дозволяють організувати багаторівневу обробку даних і широко використовуються під час роботи з таблицями значень, матрицями та іншими структурованими даними. Обчислення сум і добутків є типовими задачами програмування, що реалізуються за допомогою змінних-накопичувачів. Табулювання функцій дозволяє отримувати таблиці значень математичних залежностей і використовується у наукових та технічних обчисленнях. Обробка послідовностей даних дає змогу аналізувати великі обсяги інформації та виконувати різні операції над елементами послідовності. Розуміння цих алгоритмічних підходів є важливим етапом формування практичних навичок програмування мовами C і C++.

3.4 Ефективність алгоритмів та оцінювання їх обчислювальної складності

Під час розробки програмного забезпечення важливе значення має не лише правильність алгоритму, але й його ефективність. Один і той самий результат може бути отриманий різними алгоритмами, проте вони можуть суттєво відрізнятися за швидкістю виконання та обсягом використаної пам’яті. У задачах обробки великих обсягів даних або у системах реального часу ці відмінності можуть бути критичними. Саме тому в інформатиці важливим напрямом дослідження є аналіз ефективності алгоритмів.

Ефективність алгоритму характеризує витрати ресурсів комп’ютера, необхідні для його виконання. До основних ресурсів, що використовуються під час виконання алгоритму, належать процесорний час та обсяг оперативної пам’яті. У більшості випадків основна увага приділяється саме часовим витратам, тобто кількості операцій, які повинен виконати процесор для отримання результату.

Для оцінювання ефективності алгоритмів використовується поняття кількості операцій. Під операціями розуміють елементарні дії, що виконуються процесором. До таких

дій належать арифметичні операції, операції порівняння, присвоювання значень змінним, доступ до пам’яті та інші базові інструкції.

Кожен алгоритм складається з певної послідовності таких операцій. Під час виконання алгоритму деякі операції виконуються один раз, а інші можуть повторюватися багаторазово, наприклад у циклах. Саме повторювані операції найчастіше визначають загальну складність алгоритму.

Аналіз кількості операцій дозволяє оцінити, скільки часу приблизно знадобиться для виконання алгоритму. При цьому важливо розуміти, що точний час виконання програми залежить від багатьох факторів, зокрема від швидкодії процесора, обсягу оперативної пам’яті та особливостей компілятора. Тому під час аналізу алгоритмів зазвичай не визначають точний час виконання, а оцінюють кількість виконуваних операцій.

Під час оцінювання ефективності алгоритму важливу роль відіграє розмір вхідних даних. У більшості задач час виконання алгоритму залежить від кількості елементів, які необхідно обробити. Наприклад, якщо алгоритм працює з масивом даних, то збільшення кількості елементів масиву зазвичай призводить до збільшення кількості операцій.

Для опису залежності між розміром вхідних даних і кількістю операцій використовується поняття обчислювальної складності алгоритму. Обчислювальна складність показує, як змінюється кількість операцій при збільшенні обсягу вхідних даних.

Одним із найважливіших понять у цьому контексті є часова складність алгоритму. Часова складність визначає кількість елементарних операцій, необхідних для виконання алгоритму, як функцію від розміру вхідних даних. Наприклад, якщо алгоритм обробляє масив із n елементів і для кожного елемента виконує одну операцію, загальна кількість операцій буде пропорційна значенню n.

Під час аналізу складності алгоритмів зазвичай використовують асимптотичні оцінки. Такі оцінки дозволяють визначити характер зростання кількості операцій при значному збільшенні обсягу даних. Найбільш поширеним способом запису таких оцінок є використання спеціального математичного позначення, яке називається нотацією великого O.

Нотація великого O використовується для опису верхньої межі складності алгоритму. Вона показує, як швидко зростає кількість операцій у залежності від розміру вхідних даних. Наприклад, якщо алгоритм має складність O(n), це означає, що кількість операцій зростає пропорційно кількості елементів.

Існує кілька типових класів складності алгоритмів. Найпростішими є алгоритми з постійною складністю. У цьому випадку кількість операцій не залежить від розміру вхідних даних. Такі алгоритми виконуються за приблизно однаковий час незалежно від кількості елементів.

Наступним типом є алгоритми з лінійною складністю. У цьому випадку кількість операцій пропорційна розміру вхідних даних. Наприклад, якщо потрібно послідовно переглянути всі елементи масиву, кількість операцій буде зростати лінійно зі збільшенням кількості елементів.

Більш складними є алгоритми з квадратичною складністю. У таких алгоритмах кількість операцій пропорційна квадрату кількості елементів. Така ситуація часто виникає під час використання вкладених циклів, коли для кожного елемента необхідно виконати додаткову обробку всіх інших елементів.

Існують також алгоритми з логарифмічною складністю, експоненціальною складністю та іншими типами залежностей. Алгоритми з логарифмічною складністю зазвичай

використовуються у задачах пошуку та сортування. Експоненціальна складність характерна для деяких задач комбінаторної оптимізації.

Порівняння алгоритмів за складністю дозволяє визначити, який алгоритм є більш ефективним для розв'язання певної задачі. Наприклад, якщо один алгоритм має лінійну складність, а інший — квадратичну, то при великому обсязі даних перший алгоритм буде працювати значно швидше.

Розглянемо простий приклад. Нехай існує задача пошуку певного значення у масиві даних. Один із можливих алгоритмів полягає у послідовному перегляді всіх елементів масиву. У цьому випадку у найгіршому випадку необхідно перевірити всі елементи масиву. Кількість операцій буде пропорційна значенню n, де n — кількість елементів.

Інший алгоритм може використовувати більш ефективний підхід, наприклад поділ масиву на частини та послідовне звуження області пошуку. У такому випадку кількість операцій може зростати значно повільніше, наприклад пропорційно логарифму від кількості елементів.

Під час аналізу ефективності алгоритмів часто розглядають три основні ситуації: найкращий випадок, середній випадок та найгірший випадок. Найкращий випадок відповідає ситуації, коли алгоритм виконує мінімальну кількість операцій. Найгірший випадок відповідає максимальній кількості операцій, необхідних для виконання алгоритму.

Найгірший випадок зазвичай має найбільше значення під час аналізу алгоритмів, оскільки він гарантує верхню межу часу виконання. Якщо алгоритм працює достатньо швидко навіть у найгіршому випадку, це означає, що він буде ефективним у більшості практичних ситуацій.

Окрім часової складності, важливе значення має також просторова складність алгоритму. Просторова складність визначає обсяг пам'яті, необхідний для виконання алгоритму. У деяких задачах використання додаткової пам'яті дозволяє зменшити час виконання алгоритму.

Наприклад, алгоритм може зберігати проміжні результати обчислень у пам'яті, щоб уникнути повторного виконання тих самих операцій. Такий підхід дозволяє прискорити роботу програми, але потребує використання додаткових ресурсів пам'яті.

Під час розробки програмного забезпечення часто доводиться шукати компроміс між часовою та просторовою ефективністю алгоритму. У деяких випадках доцільно використовувати більше пам'яті для прискорення обчислень, а у інших — навпаки, зменшувати використання пам'яті за рахунок збільшення кількості обчислень.

Ще одним важливим аспектом є практична оцінка ефективності алгоритмів. Після реалізації алгоритму програміст може виконати експериментальні вимірювання часу виконання програми для різних обсягів вхідних даних. Такі вимірювання дозволяють оцінити реальну швидкодію алгоритму у конкретному програмному середовищі.

Водночас експериментальні вимірювання не завжди дають повну картину ефективності алгоритму. Саме тому теоретичний аналіз складності алгоритмів є важливим інструментом у програмуванні.

Таким чином, поняття ефективності алгоритму відіграє важливу роль у розробці програмного забезпечення. Ефективність визначається витратами ресурсів комп'ютера, необхідними для виконання алгоритму. Основним показником ефективності є кількість елементарних операцій, що виконуються під час роботи алгоритму. Аналіз залежності між кількістю операцій і розміром вхідних даних дозволяє визначити обчислювальну складність

алгоритму. Порівняння алгоритмів за складністю дає змогу обрати найбільш ефективний спосіб розв'язання задачі. Розуміння принципів аналізу алгоритмів є важливим етапом підготовки фахівців у галузі програмування та розробки програмних систем.

ТЕМА 4. ФУНКЦІЇ ТА МОДУЛЬНА ОРГАНІЗАЦІЯ ПРОГРАМ

4.1 Функції у мовах C/C++

Під час розробки програмного забезпечення важливою вимогою є структурованість програмного коду. Складні програми складаються з великої кількості інструкцій, які реалізують різні частини алгоритму. Якщо всі інструкції записати у вигляді одного великого блоку, така програма буде складною для розуміння, тестування та модифікації. Для розв'язання цієї проблеми у програмуванні використовують функції. Функції дозволяють розділити програму на логічно завершені частини, кожна з яких виконує певну задачу.

Функція — це іменований блок програмного коду, призначений для виконання певної операції або групи операцій. Після створення функції її можна викликати у різних частинах програми, передаючи їй необхідні дані та отримуючи результат виконання. Використання функцій значно підвищує зрозумілість програмного коду, сприяє повторному використанню алгоритмів і полегшує супровід програмного забезпечення.

Кожна функція має власне ім'я, список параметрів та тип значення, яке вона може повертати. Ім'я функції використовується для її виклику у програмі. Тип значення визначає, який результат повертає функція після завершення виконання. Якщо функція не повертає значення, використовується спеціальний тип void.

У програмі мовою C або C++ завжди існує одна спеціальна функція, з якої починається виконання програми. Це функція main. Саме вона є точкою входу у програму. Інші функції можуть викликатися з функції main або з інших функцій.

Під час використання функцій у програмі розрізняють кілька важливих елементів: прототип функції, опис функції та виклик функції. Кожен із цих елементів відіграє важливу роль у процесі компіляції та виконання програми.

Прототип функції — це оголошення функції без її реалізації. Прототип містить інформацію про тип значення, яке повертає функція, її ім'я та типи параметрів. Основним призначенням прототипу є повідомлення компілятора про існування функції до моменту її використання у програмі.

У мовах C і C++ компілятор обробляє програму послідовно зверху вниз. Якщо функція викликається раніше, ніж описано її реалізацію, компілятор повинен знати її тип та список параметрів. Саме для цього використовуються прототипи функцій.

Прототип функції зазвичай розміщується на початку програмного файлу або у спеціальних заголовкових файлах. Це дозволяє використовувати функцію у різних частинах програми без необхідності розміщувати її реалізацію перед кожним викликом.

Опис функції містить повну реалізацію функції. Він складається із заголовка функції та її тіла. Заголовок функції містить тип значення, яке повертає функція, її ім'я та список параметрів. Після заголовка розміщується тіло функції, яке містить програмні інструкції, що виконуються під час її виклику.

Тіло функції записується у фігурних дужках і може містити оголошення змінних, арифметичні обчислення, виклики інших функцій, умовні оператори та цикли. Після

завершення виконання інструкцій функція може повернути певне значення за допомогою спеціального оператора повернення.

Виклик функції — це інструкція, за допомогою якої програма передає керування функції для виконання. Під час виклику функції можуть передаватися значення параметрів. Після завершення роботи функції виконання програми продовжується з тієї точки, де був виконаний виклик.

Використання функцій дозволяє організувати програму у вигляді набору взаємопов'язаних модулів. Кожна функція виконує окрему задачу, що робить програму більш зрозумілою та зручною для розробки.

Під час виклику функції важливу роль відіграють параметри. Параметри використовуються для передавання даних між функціями. У програмуванні розрізняють формальні та фактичні параметри.

Формальні параметри — це змінні, які оголошуються у заголовку функції. Вони використовуються всередині функції для роботи з переданими даними. Формальні параметри існують лише у межах функції і не доступні поза її тілом.

Фактичні параметри — це значення або змінні, які передаються функції під час її виклику. Фактичні параметри можуть бути змінними, константами або результатами обчислення певних виразів.

Під час виклику функції значення фактичних параметрів передаються формальним параметрам. Кількість і типи фактичних параметрів повинні відповідати параметрам, визначеним у прототипі функції.

У мовах C і C++ існують різні способи передавання параметрів до функції. Найбільш поширеними є передавання параметрів за значенням та передавання параметрів за адресою.

Передавання параметрів за значенням означає, що під час виклику функції значення фактичного параметра копіюється у відповідний формальний параметр. У цьому випадку функція працює з копією значення, а не з оригінальною змінною.

Це означає, що зміни, виконані всередині функції, не впливають на значення змінної у викликаючій програмі. Такий підхід забезпечує безпечність роботи функцій, оскільки виклик функції не змінює значення змінних у головній програмі.

Передавання параметрів за значенням використовується у більшості випадків, коли функція повинна лише використовувати значення параметра для виконання обчислень.

Іншим способом є передавання параметрів за адресою. У цьому випадку функції передається не саме значення змінної, а адреса її розміщення у пам'яті. Це дозволяє функції безпосередньо працювати з тією самою змінною, яка використовується у викликаючій програмі.

Під час передавання параметрів за адресою функція може змінювати значення змінної у головній програмі. Такий підхід використовується у тих випадках, коли функція повинна повернути кілька результатів або змінити значення змінних, переданих їй як параметри.

Передавання параметрів за адресою широко використовується під час роботи з масивами, великими структурами даних або у функціях, що виконують обробку даних без створення додаткових копій.

Вибір способу передавання параметрів залежить від задачі, яку виконує функція. Передавання за значенням є більш безпечним і зрозумілим, тоді як передавання за адресою дозволяє реалізувати більш гнучкі алгоритми обробки даних.

Під час розробки програм важливо чітко розуміти різницю між цими способами передавання параметрів. Неправильний вибір способу передавання може призвести до помилок у роботі програми або до неочікуваної зміни значень змінних.

Використання функцій також сприяє повторному використанню програмного коду. Одна й та сама функція може використовуватися у різних частинах програми або навіть у різних програмах. Це дозволяє значно скоротити обсяг коду та підвищити ефективність розробки програмного забезпечення.

Крім того, функції полегшують тестування програм. Кожну функцію можна перевіряти окремо, що дозволяє швидше знаходити та виправляти помилки. Такий підхід є основою модульного програмування.

Таким чином, функції є важливим елементом програмування мовами C і C++. Вони дозволяють розділяти програму на логічно завершені частини та організовувати програмний код у вигляді взаємопов'язаних модулів. Основними елементами використання функцій є прототип функції, її опис та виклик. Під час роботи з функціями використовуються формальні та фактичні параметри, які забезпечують передавання даних між частинами програми. Передавання параметрів може виконуватися за значенням або за адресою, що визначає спосіб взаємодії функції з даними. Розуміння цих принципів є важливим етапом формування практичних навичок програмування та створення структурованих програмних систем.

4.2 Область видимості змінних та організація пам'яті

Під час виконання програм мовами C і C++ комп'ютер використовує оперативну пам'ять для зберігання змінних, параметрів функцій та службової інформації, необхідної для виконання алгоритму. Організація використання пам'яті є важливою частиною роботи будь-якої програми. Розуміння того, як створюються змінні, де вони зберігаються та як відбувається передавання керування між функціями, дозволяє програмісту ефективніше розробляти програмний код і уникати помилок.

Одним із ключових понять у цьому контексті є область видимості змінних. Область видимості визначає частину програми, у межах якої певна змінна може бути використана. У мовах C і C++ змінні можуть мати локальну або глобальну область видимості. Вибір області видимості впливає на доступність змінної, тривалість її існування у пам'яті та спосіб взаємодії з іншими частинами програми.

Глобальні змінні оголошуються поза межами функцій, зазвичай на початку програмного файлу. Такі змінні доступні для використання у всіх функціях програми. Оскільки глобальні змінні оголошуються поза функціями, вони створюються ще до початку виконання програми і зберігаються у пам'яті протягом усього часу її роботи.

Глобальні змінні можуть використовуватися для зберігання даних, які повинні бути доступними у різних частинах програми. Наприклад, у деяких задачах може виникнути потреба зберігати загальні параметри системи або результати обчислень, які використовуються кількома функціями.

Водночас використання глобальних змінних має певні недоліки. Оскільки такі змінні доступні з будь-якої частини програми, їх значення може змінюватися різними функціями. Це ускладнює контроль над програмним кодом і може призвести до помилок, які важко

знайти. Саме тому у структурованому програмуванні рекомендується обмежувати використання глобальних змінних і віддавати перевагу локальним змінним.

Локальні змінні оголошуються всередині функцій або окремих блоків коду. Такі змінні доступні лише у межах тієї функції або блоку, у якому вони оголошені. Поза межами цього блоку доступ до локальної змінної неможливий.

Локальні змінні створюються під час виклику функції і знищуються після завершення її виконання. Це означає, що кожного разу під час виклику функції створюється новий набір локальних змінних. Такий механізм дозволяє різним викликам функції працювати незалежно один від одного.

Використання локальних змінних є важливим принципом структурованого програмування. Воно дозволяє обмежити область доступу до даних і зробити програму більш зрозумілою. Кожна функція працює зі своїм власним набором змінних, що зменшує ймовірність випадкової зміни даних у інших частинах програми.

Під час виконання функцій важливу роль відіграє механізм організації пам’яті. У сучасних комп’ютерних системах пам’ять програми поділяється на кілька логічних областей. Однією з таких областей є стек викликів.

Стек викликів — це спеціальна структура пам’яті, яка використовується для зберігання інформації про активні виклики функцій. Стек працює за принципом останній прийшов — перший вийшов. Це означає, що остання викликана функція завершується першою.

Коли програма викликає функцію, у стек викликів записується спеціальний блок інформації, який називають кадром стеку або записом активації функції. Цей блок містить інформацію, необхідну для виконання функції. До такої інформації належать значення параметрів функції, локальні змінні, а також адреса повернення.

Адреса повернення — це місце у програмі, з якого була викликана функція. Після завершення виконання функції програма повертається саме до цієї адреси, щоб продовжити виконання наступних інструкцій.

Коли функція завершує свою роботу, відповідний кадр видаляється зі стеку. У результаті відновлюється стан попередньої функції, і виконання програми продовжується.

Стек викликів дозволяє програмі коректно працювати з великою кількістю вкладених викликів функцій. Наприклад, одна функція може викликати іншу, та — ще одну, і так далі. Для кожного такого виклику у стеку створюється окремий кадр.

Особливо важливу роль стек викликів відіграє у рекурсивних функціях. Рекурсія виникає у випадку, коли функція викликає саму себе. Під час кожного рекурсивного виклику у стеку створюється новий кадр, що містить власні параметри та локальні змінні.

Якщо кількість рекурсивних викликів стає надто великою, стек може переповнитися. Така ситуація називається переповненням стеку. Саме тому під час використання рекурсії необхідно забезпечити наявність умови завершення рекурсивного процесу.

Під час виконання функції важливу роль відіграє оператор return. Цей оператор використовується для завершення роботи функції та повернення керування до тієї частини програми, з якої була викликана функція.

Оператор return може також повертати певне значення. У цьому випадку значення, зазначене після оператора return, передається викликаючій функції як результат виконання. Тип значення, що повертається, повинен відповідати типу, визначеному у заголовку функції.

Наприклад, якщо функція оголошена як така, що повертає ціле число, оператор return повинен повертати значення цілого типу. Якщо ж функція не повинна повертати значення,

використовується тип void, і оператор return може використовуватися лише для завершення виконання функції без повернення результату.

Після виконання оператора return подальші інструкції у тілі функції не виконуються. Це означає, що оператор return завершує роботу функції негайно. Саме тому його часто використовують для завершення функції після виконання певних умов.

У деяких випадках функція може містити кілька операторів return. Це дозволяє завершувати виконання функції у різних точках залежно від результатів перевірки умов. Такий підхід може спростити структуру алгоритму, однак потребує уважного проєктування програмного коду.

Під час використання оператора return важливо пам’ятати, що після завершення функції її локальні змінні знищуються. Саме тому функція не повинна повертати посилання на локальні змінні, оскільки після завершення функції вони більше не існують у пам’яті.

Організація пам’яті під час виконання функцій є важливою частиною роботи будь-якої програми. Розуміння того, як створюються та знищуються змінні, як працює стек викликів і як відбувається передавання керування між функціями, дозволяє програмісту краще контролювати роботу програмного коду.

Особливо важливо це під час розробки великих програмних систем, у яких функції можуть викликатися багаторазово і взаємодіяти одна з одною. У таких системах правильна організація областей видимості змінних і ефективне використання пам’яті дозволяють створювати надійні та ефективні програми.

Таким чином, у мовах C і C++ змінні можуть мати локальну або глобальну область видимості. Глобальні змінні доступні у всій програмі та існують протягом усього часу її виконання, тоді як локальні змінні створюються під час виклику функції і знищуються після її завершення. Під час виконання функцій використовується спеціальна структура пам’яті — стек викликів, яка зберігає інформацію про активні виклики функцій, їх параметри та локальні змінні. Оператор return використовується для завершення виконання функції та повернення керування до викликаючої частини програми. Розуміння цих механізмів є важливою основою для подальшого вивчення програмування та розробки складних програмних систем.

4.3 Модульна побудова програм та організація розробки програмного коду

У процесі створення програмного забезпечення програміст стикається з необхідністю реалізації складних алгоритмів, що складаються з великої кількості взаємопов’язаних операцій. Якщо реалізовувати таку програму у вигляді одного великого блоку інструкцій, її структура стає складною для розуміння, перевірки та модифікації. Саме тому у сучасному програмуванні широко застосовується принцип модульної побудови програм. Цей принцип передбачає розділення програми на окремі частини, які називаються модулями. Кожен модуль реалізує певну функціональність і може розроблятися, перевірятися та використовуватися незалежно від інших частин програми.

Модульна організація програмного забезпечення є одним із фундаментальних принципів програмної інженерії. Вона дозволяє значно спростити процес розробки, зробити програмний код більш зрозумілим та полегшити його подальшу підтримку. У мовах програмування C і C++ модульність зазвичай реалізується за допомогою функцій, окремих файлів програмного коду та бібліотек.

Основною ідеєю модульної побудови є поділ складної задачі на менші частини. Такий підхід називається декомпозицією задачі. Декомпозиція дозволяє розбити складну проблему на набір підзадач, кожна з яких може бути розв'язана окремо. Після цього результати розв'язання підзадач об'єднуються для отримання кінцевого результату.

Під час розробки алгоритму програміст спочатку аналізує загальну структуру задачі. Далі визначаються основні етапи її розв'язання. Кожен із таких етапів може бути реалізований окремою функцією або модулем. Такий підхід дозволяє створювати програму поступово, реалізуючи і перевіряючи окремі її частини.

Декомпозиція задачі є важливим інструментом алгоритмізації. Вона дозволяє зменшити складність розробки програмного забезпечення, оскільки програміст працює не з усією задачею одразу, а з її окремими компонентами. Кожна підзадача зазвичай має чітко визначені вхідні дані та результат виконання.

Наприклад, якщо необхідно створити програму для обробки числових даних, загальну задачу можна поділити на кілька підзадач. Одна функція може виконувати введення даних, інша — їх обробку, а третя — виведення результатів. Такий підхід дозволяє чітко структурувати програму та полегшити її подальший розвиток.

Однією з важливих переваг модульної побудови програм є можливість повторного використання коду. Повторне використання коду означає застосування вже створених програмних компонентів у різних частинах програми або навіть у різних програмних проєктах. Це дозволяє значно скоротити час розробки програмного забезпечення та підвищити його надійність.

Функції, що реалізують певні універсальні алгоритми, можуть використовуватися багаторазово. Наприклад, функція обчислення математичного виразу або функція сортування масиву може застосовуватися у різних програмах. Якщо така функція вже перевірена та протестована, її використання дозволяє уникнути повторного написання коду.

Повторне використання коду є одним із ключових принципів сучасної програмної інженерії. У великих програмних системах часто створюються бібліотеки функцій, які містять готові програмні компоненти для виконання типових задач. Використання таких бібліотек значно спрощує процес розробки програм.

У мовах C і C++ повторне використання коду часто реалізується за допомогою заголовкових файлів та бібліотек. Заголовкові файли містять оголошення функцій, тоді як їх реалізація може знаходитися в інших програмних файлах. Така структура дозволяє використовувати одні й ті самі функції у різних частинах програми.

Ще однією важливою перевагою модульної побудови є можливість незалежного тестування окремих частин програми. Кожен модуль можна перевіряти окремо від інших компонентів системи. Це значно спрощує процес пошуку помилок і підвищує надійність програмного забезпечення.

Тестування програм є важливим етапом розробки програмного забезпечення. Метою тестування є перевірка правильності роботи програми та виявлення можливих помилок. Під час тестування перевіряється відповідність результатів виконання програми заданим вимогам.

Тестування може виконуватися на різних етапах розробки програмного забезпечення. Спочатку перевіряються окремі функції або модулі. Такий підхід називається модульним тестуванням. Модульне тестування дозволяє перевірити правильність роботи кожної функції незалежно від інших частин програми.

Після перевірки окремих модулів виконується інтеграційне тестування. На цьому етапі перевіряється правильність взаємодії між різними частинами програми. Це дозволяє виявити помилки, які можуть виникати під час обміну даними між функціями.

Завершальним етапом є системне тестування. Під час системного тестування перевіряється робота всієї програмної системи в цілому. Це дозволяє переконатися, що всі компоненти програми працюють разом правильно і забезпечують очікуваний результат.

Під час тестування використовуються різні набори вхідних даних. Зокрема, перевіряються типові випадки використання програми, граничні значення параметрів та некоректні дані. Такий підхід дозволяє перевірити стійкість програми до різних ситуацій.

Після виявлення помилок виконується процес налагодження програм. Налагодження — це процес пошуку та виправлення помилок у програмному коді. Помилки у програмі можуть виникати з різних причин, наприклад через неправильну реалізацію алгоритму, помилки у логіці умов або неправильну роботу зі змінними.

Під час налагодження програміст аналізує поведінку програми та визначає місце виникнення помилки. Для цього можуть використовуватися різні методи, зокрема виведення проміжних результатів, покрокове виконання програми та використання спеціальних інструментів налагодження.

Сучасні середовища розробки програмного забезпечення містять вбудовані засоби налагодження. Такі засоби дозволяють виконувати програму крок за кроком, переглядати значення змінних та аналізувати послідовність виконання інструкцій.

Важливою частиною налагодження є аналіз логіки роботи програми. У деяких випадках помилка може виникати не через синтаксичну помилку, а через неправильну реалізацію алгоритму. Саме тому програміст повинен уважно перевіряти логіку виконання програми.

Ще одним важливим аспектом розробки програмного забезпечення є документування програмного коду. Документування передбачає додавання коментарів до програми, які пояснюють призначення функцій, змінних та окремих фрагментів алгоритму. Це значно полегшує розуміння програми іншими розробниками та спрощує її подальшу модифікацію.

Таким чином, принципи модульної побудови програм відіграють важливу роль у сучасному програмуванні. Розділення програми на окремі модулі дозволяє спростити процес розробки та зробити програмний код більш зрозумілим і зручним для супроводу. Декомпозиція задачі на підзадачі дозволяє поступово реалізовувати складні алгоритми. Повторне використання коду сприяє скороченню часу розробки та підвищенню надійності програмного забезпечення. Тестування і налагодження програм дозволяють виявляти та виправляти помилки, забезпечуючи правильну роботу програмних систем.

ТЕМА 5. МАСИВИ ТА БАЗОВІ АЛГОРИТМИ ОБРОБКИ ДАНИХ

5.1 Масиви у мовах C/C++

Під час розробки програм часто виникає необхідність обробляти велику кількість однотипних даних. Наприклад, програма може працювати з набором чисел, результатами вимірювань, значеннями функції або іншими послідовностями інформації. Використання окремих змінних для кожного елемента таких даних є незручним і ускладнює структуру програми. Для розв’язання цієї проблеми у програмуванні використовуються масиви. Масив

дозволяє зберігати багато значень одного типу під спільним ім’ям, що значно спрощує обробку послідовностей даних.

Масив — це впорядкована сукупність елементів однакового типу, що зберігаються у послідовних комірках пам’яті. Кожен елемент масиву має свій порядковий номер, який називається індексом. За допомогою індексу можна отримати доступ до будь-якого елемента масиву. У мовах C і C++ нумерація елементів масиву починається з нуля. Це означає, що перший елемент масиву має індекс 0, другий — індекс 1 і так далі.

Використання масивів дозволяє значно спростити реалізацію алгоритмів обробки даних. Замість роботи з великою кількістю окремих змінних програміст використовує одну структуру даних і звертається до її елементів за допомогою індексів. Це особливо зручно під час використання циклів, коли одна й та сама операція виконується для всіх елементів масиву.

Найпростішим типом масиву є одновимірний масив. Одновимірний масив можна розглядати як послідовність елементів, розташованих один за одним. Така структура даних нагадує ряд значень, де кожне значення має свій номер. Одновимірні масиви широко використовуються у програмуванні для зберігання наборів чисел, символів або інших даних.

Перш ніж використовувати масив у програмі, його необхідно оголосити. Оголошення масиву передбачає визначення типу елементів масиву, його імені та кількості елементів. Під час оголошення масиву компілятор виділяє відповідний обсяг пам’яті для зберігання всіх його елементів.

Тип елементів масиву визначає, які значення можуть зберігатися у цьому масиві. Наприклад, якщо масив оголошений як масив цілих чисел, усі його елементи повинні мати цілий тип. Якщо ж необхідно працювати з дійсними числами, використовується масив відповідного типу.

Кількість елементів масиву визначається під час його оголошення і називається розміром масиву. У мовах C і C++ розмір масиву повинен бути визначений під час компіляції програми. Це означає, що програміст повинен заздалегідь визначити, скільки елементів буде містити масив.

Після оголошення масиву його елементи можуть отримувати початкові значення. Цей процес називається ініціалізацією масиву. Ініціалізація може виконуватися під час оголошення або у процесі виконання програми.

Під час ініціалізації масиву програміст може задати значення для всіх або для деяких його елементів. Якщо значення елементів не задані явно, їх початковий стан може бути невизначеним. Саме тому під час розробки програм рекомендується виконувати явну ініціалізацію масивів.

Після оголошення та ініціалізації масиву програма може виконувати різні операції над його елементами. Найпоширенішими операціями є введення даних у масив, обробка значень елементів та виведення результатів.

Введення елементів масиву зазвичай виконується за допомогою циклу. Під час кожної ітерації циклу програма зчитує значення для одного елемента масиву. Такий підхід дозволяє послідовно заповнювати всі елементи масиву.

Циклічна обробка масивів є одним із найбільш поширених прийомів програмування. Використовуючи цикл, програміст може виконувати однакові операції над усіма елементами масиву. Наприклад, можна обчислити суму всіх елементів, знайти максимальне або мінімальне значення або виконати інші математичні операції.

Виведення елементів масиву також зазвичай виконується за допомогою циклу. Під час кожної ітерації програма звертається до чергового елемента масиву і відображає його значення на екрані. Такий підхід дозволяє легко формувати таблиці значень або інші структури даних.

Під час роботи з масивами важливо правильно використовувати індекси. Індекс повинен знаходитися у межах допустимого діапазону. Якщо програма звертається до елемента масиву з індексом, що виходить за межі визначеного розміру масиву, може виникнути помилка доступу до пам’яті.

Саме тому програміст повинен уважно контролювати значення індексів під час роботи з масивами. Найчастіше для цього використовуються цикли, у яких межі зміни змінної-індексу відповідають розміру масиву.

У більш складних програмах масиви часто передаються у функції для подальшої обробки. Передавання масивів у функції дозволяє реалізувати модульну структуру програмного коду. Одна функція може відповідати за введення даних, інша — за їх обробку, а третя — за виведення результатів.

Під час передавання масиву у функцію важливо розуміти особливості роботи з пам’яттю. У мовах C і C++ масив передається у функцію не як повна копія даних, а як адреса першого елемента масиву. Це означає, що функція отримує доступ до тих самих даних, які знаходяться у головній програмі.

Такий підхід має важливу перевагу. Передавання адреси масиву дозволяє уникнути створення копії всіх його елементів. Це особливо важливо у випадках, коли масив містить велику кількість даних. Передавання лише адреси значно зменшує витрати пам’яті та прискорює виконання програми.

Водночас передавання масиву за адресою означає, що функція може змінювати значення його елементів. Усі зміни, виконані всередині функції, будуть відображені у головній програмі. Саме тому під час розробки програм необхідно уважно контролювати операції, які виконуються над елементами масиву.

Під час передавання масиву у функцію зазвичай додатково передається його розмір. Це необхідно для того, щоб функція могла правильно організувати обробку елементів масиву. Без інформації про розмір масиву функція не може визначити кількість елементів, які потрібно обробити.

У деяких випадках функції можуть працювати лише з частиною масиву. Наприклад, функція може обробляти лише перші кілька елементів або певний діапазон індексів. Такий підхід дозволяє гнучко організовувати роботу з великими масивами даних.

Масиви широко використовуються у різних галузях програмування. Вони застосовуються під час обробки результатів експериментів, статистичного аналізу даних, роботи з таблицями значень та реалізації різних алгоритмів обчислень. Завдяки простій структурі та ефективному використанню пам’яті масиви є одним із основних інструментів роботи з даними у програмуванні.

Таким чином, масив є структурою даних, яка дозволяє зберігати послідовність однотипних значень у пам’яті комп’ютера. Одновимірні масиви представляють собою впорядковану послідовність елементів, доступ до яких здійснюється за допомогою індексів. Перед використанням масиву його необхідно оголосити та, за потреби, ініціалізувати. Введення і виведення елементів масиву зазвичай виконується за допомогою циклів. Масиви можуть передаватися у функції для подальшої обробки, що дозволяє реалізувати модульну

структуру програмного забезпечення. Розуміння принципів роботи з масивами є важливою складовою підготовки фахівців у галузі програмування.

5.2 Алгоритми обробки масивів: пошук, сортування та оцінювання ефективності

Під час роботи з масивами даних однією з основних задач є їх обробка. Масиви дозволяють зберігати велику кількість однотипних елементів, а алгоритми обробки масивів дають змогу виконувати різні операції над цими даними. До найпоширеніших задач належать пошук певного елемента у масиві, впорядкування елементів за зростанням або спаданням, визначення максимальних і мінімальних значень, а також виконання статистичних обчислень. У програмуванні розроблено велику кількість алгоритмів для роботи з масивами, однак у початковому курсі алгоритмізації особливу увагу приділяють базовим методам пошуку і сортування.

Алгоритми обробки масивів зазвичай реалізуються за допомогою циклів. Циклічні конструкції дозволяють послідовно переглядати всі елементи масиву і виконувати необхідні операції. У більшості випадків програма виконує однакові дії для кожного елемента масиву, змінюючи лише значення індексу.

Однією з найпростіших задач під час роботи з масивами є пошук елемента. Пошук передбачає визначення позиції елемента у масиві, значення якого відповідає певному критерію. Наприклад, може виникнути потреба знайти елемент із заданим значенням або визначити перший елемент, що задовольняє певну умову.

Найпростішим методом пошуку є лінійний пошук. Цей алгоритм полягає у послідовному перегляді всіх елементів масиву. Програма починає перевірку з першого елемента і порівнює його значення з шуканим значенням. Якщо значення збігаються, пошук завершується, і програма повідомляє індекс знайденого елемента.

Якщо значення не збігається, програма переходить до наступного елемента і повторює перевірку. Такий процес продовжується до тих пір, поки елемент не буде знайдено або поки не буде переглянуто всі елементи масиву.

Лінійний пошук є простим для реалізації і не потребує попереднього впорядкування масиву. Саме тому цей алгоритм часто використовується у навчальних задачах та у ситуаціях, коли масив має невеликий розмір.

Водночас ефективність лінійного пошуку залежить від кількості елементів масиву. У найгіршому випадку програма повинна перевірити всі елементи масиву, щоб переконатися, що шукане значення відсутнє. Таким чином, кількість операцій у цьому алгоритмі зростає пропорційно розміру масиву.

Ще однією важливою задачею під час роботи з масивами є сортування. Сортування передбачає упорядкування елементів масиву за певним правилом. Найчастіше використовується сортування за зростанням або за спаданням значень.

Сортування є важливою операцією обробки даних. Упорядковані дані значно легше аналізувати, а деякі алгоритми пошуку можуть працювати значно швидше, якщо масив попередньо відсортований.

Існує велика кількість алгоритмів сортування. У початковому курсі програмування зазвичай розглядаються прості алгоритми сортування, які легко зрозуміти та реалізувати. До таких алгоритмів належать сортування вибором, сортування обміном та сортування вставками.

Одним із найпростіших алгоритмів є сортування вибором. Ідея цього алгоритму полягає у послідовному виборі найменшого елемента з невідсортованої частини масиву та переміщенні його у відповідну позицію.

На першому етапі програма знаходить найменший елемент у всьому масиві та міняє його місцями з першим елементом. Після цього перший елемент вважається відсортованим. На наступному етапі програма шукає найменший елемент серед решти елементів і розміщує його на другій позиції. Такий процес повторюється до тих пір, поки весь масив не буде впорядкований.

Сортування вибором є простим для реалізації і добре ілюструє принципи роботи алгоритмів сортування. Однак цей алгоритм не є дуже ефективним для великих масивів, оскільки потребує великої кількості порівнянь.

Ще одним поширеним алгоритмом є сортування обміном. Цей метод також відомий як бульбашкове сортування. Його принцип полягає у послідовному порівнянні сусідніх елементів масиву та їх обміні у випадку неправильного порядку.

Під час кожного проходу по масиву найбільший елемент поступово переміщується до кінця масиву. Після кількох повторень цього процесу всі елементи масиву займають правильні позиції.

Бульбашкове сортування є простим для реалізації, проте його ефективність також є невисокою. Кількість операцій у цьому алгоритмі швидко зростає зі збільшенням розміру масиву.

Ще одним простим методом сортування є сортування вставками. У цьому алгоритмі масив умовно поділяється на відсортовану та невідсортовану частини. Спочатку відсортованою вважається лише перша позиція масиву. Далі кожен наступний елемент вставляється у відповідне місце відсортованої частини.

Під час виконання цього алгоритму програма порівнює новий елемент з елементами відсортованої частини і переміщує його у правильну позицію. Такий підхід нагадує процес сортування гральних карт у руці.

Сортування вставками є більш ефективним у випадках, коли масив уже частково впорядкований. Саме тому цей алгоритм іноді використовується у практичних задачах.

Під час вибору алгоритму сортування важливо враховувати його ефективність. Ефективність алгоритму визначається кількістю операцій, необхідних для виконання сортування.

Для порівняння алгоритмів використовують поняття обчислювальної складності. Обчислювальна складність показує, як змінюється кількість операцій залежно від розміру масиву. Якщо кількість операцій швидко зростає зі збільшенням кількості елементів, алгоритм вважається менш ефективним.

Прості алгоритми сортування, такі як сортування вибором або бульбашкове сортування, мають відносно високу складність. У таких алгоритмах кількість операцій приблизно пропорційна квадрату кількості елементів масиву.

Це означає, що якщо розмір масиву збільшується у два рази, кількість операцій може зрости приблизно у чотири рази. Саме тому такі алгоритми неефективні для дуже великих масивів даних.

У більш складних програмних системах використовуються більш ефективні алгоритми сортування, наприклад швидке сортування або сортування злиттям. Однак для розуміння основ алгоритмізації достатньо вивчити прості алгоритми сортування.

Порівняння алгоритмів за ефективністю дозволяє визначити, який метод обробки даних є більш доцільним у конкретній задачі. У невеликих програмах різниця між алгоритмами може бути майже непомітною, проте під час роботи з великими масивами даних ефективність алгоритму стає критично важливою.

Таким чином, алгоритми обробки масивів відіграють важливу роль у програмуванні. Лінійний пошук дозволяє знаходити елементи у масиві шляхом послідовного перегляду всіх значень. Прості алгоритми сортування дозволяють упорядковувати елементи масиву за заданим правилом. Порівняння алгоритмів за ефективністю дає змогу визначити найбільш доцільний спосіб обробки даних залежно від розміру масиву та вимог до швидкодії програми. Розуміння цих алгоритмів є важливим етапом формування практичних навичок програмування.

5.3 Алгоритмізація як основа розробки програмного забезпечення

У процесі вивчення дисципліни алгоритмізації та програмування студенти ознайомлюються з фундаментальними принципами створення програмного забезпечення. Основою будь-якої програмної системи є алгоритм — формалізований опис послідовності дій, які необхідно виконати для розв’язання певної задачі. Саме алгоритм визначає логіку роботи програми, структуру обробки даних та взаємодію окремих компонентів програмного забезпечення. Без чітко сформульованого алгоритму створення надійної та ефективної програми практично неможливе.

Алгоритмізація є процесом розробки алгоритмів для розв’язання прикладних задач. Вона передує безпосередньому написанню програмного коду і дозволяє сформувати логічну структуру майбутньої програми. Під час алгоритмізації програміст аналізує умову задачі, визначає необхідні дані, встановлює послідовність обчислень і формує структуру алгоритму. Лише після цього алгоритм може бути реалізований мовою програмування.

Однією з головних особливостей алгоритмізації є її незалежність від конкретної мови програмування. Алгоритм описує логіку розв’язання задачі у загальному вигляді. Реалізація цього алгоритму може бути виконана різними мовами програмування або навіть різними програмними системами. Саме тому алгоритмічне мислення є важливою навичкою для будь-якого фахівця у галузі інформаційних технологій.

У процесі вивчення алгоритмізації студенти знайомляться з основними властивостями алгоритмів. До таких властивостей належать визначеність, дискретність, масовість та результативність. Визначеність означає однозначність кожної команди алгоритму. Дискретність передбачає поділ процесу обчислення на окремі кроки. Масовість означає можливість застосування алгоритму до широкого класу задач. Результативність гарантує завершення алгоритму за скінченну кількість кроків.

Алгоритми можуть подаватися різними способами. Найпростішим способом є словесний опис послідовності дій. Більш формалізованими методами є використання блок-схем або псевдокоду. Такі засоби дозволяють наочно представити структуру алгоритму і полегшують його подальшу реалізацію у вигляді програмного коду.

Будь-який алгоритм будується на основі базових алгоритмічних структур. До таких структур належать послідовність, розгалуження та повторення. Послідовність передбачає виконання команд у визначеному порядку. Розгалуження дозволяє обирати різні варіанти виконання залежно від певної умови. Повторення забезпечує багаторазове виконання певної групи інструкцій.

Ці три структури є основою побудови будь-якого алгоритму. У мовах програмування вони реалізуються за допомогою відповідних операторів. Послідовність інструкцій утворює основу програмного коду, умовні оператори реалізують розгалуження, а оператори циклу забезпечують повторення обчислень.

Після розробки алгоритму наступним етапом є його реалізація мовою програмування. Програмний код є формальним записом алгоритму у вигляді інструкцій, зрозумілих комп'ютеру. Мова програмування визначає правила запису таких інструкцій, типи даних, структури керування та інші елементи програмного коду.

У курсі алгоритмізації та програмування зазвичай використовуються мови C або C++, які широко застосовуються у системному програмуванні, розробці прикладних програм і вбудованих систем. Ці мови дозволяють ефективно реалізовувати алгоритми та забезпечують гнучке керування пам'яттю комп'ютера.

Під час написання програм програміст працює з різними типами даних. До основних типів належать цілі числа, дійсні числа, символи та логічні значення. Змінні використовуються для зберігання даних у пам'яті комп'ютера, а константи представляють значення, які не змінюються під час виконання програми.

Окрім простих змінних, у програмуванні широко використовуються складні структури даних. Однією з найважливіших структур є масив. Масив дозволяє зберігати велику кількість однотипних елементів і ефективно виконувати операції над ними. Обробка масивів часто включає пошук елементів, сортування даних та виконання статистичних обчислень.

Під час розробки програм важливо також використовувати функції. Функції дозволяють розбивати програму на окремі логічні частини, кожна з яких виконує певну задачу. Такий підхід називається модульною організацією програмного коду.

Модульна побудова програм має низку важливих переваг. По-перше, вона підвищує зрозумілість програмного коду. По-друге, дозволяє повторно використовувати окремі компоненти програми. По-третє, спрощує тестування і налагодження програмного забезпечення.

Під час виконання програм комп'ютер використовує різні області пам'яті. Однією з таких областей є стек викликів. Стек викликів використовується для зберігання інформації про активні функції, їх параметри та локальні змінні. Коли функція викликається, у стеку створюється новий запис. Після завершення виконання функції цей запис видаляється.

Правильне розуміння організації пам'яті є важливим для ефективного програмування. Воно дозволяє уникати помилок, пов'язаних із неправильним використанням змінних або некоректним керуванням ресурсами системи.

Під час розробки програм важливу роль відіграє ефективність алгоритмів. Ефективність алгоритму визначається кількістю операцій, необхідних для його виконання, а також обсягом пам'яті, який використовується програмою. Аналіз ефективності алгоритмів дозволяє порівнювати різні способи розв'язання задач і обирати найбільш оптимальний варіант.

Для оцінювання ефективності алгоритмів використовується поняття обчислювальної складності. Складність алгоритму показує, як змінюється кількість операцій залежно від розміру вхідних даних. Наприклад, деякі алгоритми мають лінійну складність, а інші — квадратичну або логарифмічну.

Вибір ефективного алгоритму є важливим завданням програміста. У задачах, пов’язаних із великими обсягами даних, ефективність алгоритму може суттєво впливати на швидкодію програмної системи.

Алгоритми є лише одним із компонентів програмного забезпечення. У реальних проєктах створення програм відбувається у межах життєвого циклу програмного забезпечення. Життєвий цикл включає всі етапи створення програмної системи — від аналізу вимог до супроводу готового програмного продукту.

Життєвий цикл програмного забезпечення зазвичай включає такі етапи: аналіз вимог, проєктування, реалізація програмного коду, тестування та супровід. На етапі аналізу вимог визначаються функціональні можливості майбутньої системи. На етапі проєктування розробляється структура програми та її алгоритми.

Після цього виконується реалізація програмного коду. Саме на цьому етапі алгоритми перетворюються на програму, яка може виконуватися комп’ютером. Наступним етапом є тестування, під час якого перевіряється правильність роботи програми. Завершальним етапом є супровід програмного забезпечення, що передбачає виправлення помилок і внесення змін у програму.

Таким чином, алгоритми, програмний код і життєвий цикл програмного забезпечення тісно пов’язані між собою. Алгоритм визначає логіку розв’язання задачі. Програмний код є реалізацією цього алгоритму мовою програмування. Життєвий цикл програмного забезпечення забезпечує організацію процесу розробки, тестування та експлуатації програмних систем.

Алгоритмізація відіграє ключову роль у всьому цьому процесі. Саме алгоритм визначає, як саме програма буде обробляти дані, які операції виконуватиме та який результат отримає користувач. Тому формування алгоритмічного мислення є однією з найважливіших цілей навчання програмуванню.

Узагальнюючи матеріал курсу, можна зробити висновок, що алгоритмізація є фундаментальною основою розробки програмного забезпечення. Розуміння принципів побудови алгоритмів, використання структур керування, робота з даними та організація програмного коду дозволяють створювати ефективні програмні системи. Ці знання є базою для подальшого вивчення програмної інженерії, розробки складних програмних продуктів та використання сучасних інформаційних технологій.

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна

1. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. 4th ed. Cambridge, MA : MIT Press, 2022. 1312 p. ISBN 9780262046305.

2. Седжвік Р., Уейн К. Алгоритми. 4-те вид. Київ : BHV, 2018. 992 с. ISBN 9789665524762.

3. Рудий Т. В., Паранчук Я. С., Сеник В. В. Алгоритмізація та програмування. Ч. 1. Структурне програмування : навч. посіб. Львів : ЛДУВС, 2023. 240 с.

4. Ярошко С. А., Ярошко О. С. Методи розробки алгоритмів. Програмування : навч. посіб. Львів : ЛНУ імені Івана Франка, 2022. 248 с.

5. Лавріщева К. М. Основи алгоритмізації та програмування. Київ : Академія, 2020. 304 с.

Допоміжна

6. Вірт Н. Алгоритми + структури даних = програми. Київ : Діалектика, 2019. 366 с.

7. Трофименко О. Г., Прокоп Ю. В., Логінова Н. І., Задерейко О. В. C++. Алгоритмізація та програмування : підручник. 2-ге вид. Одеса : Фенікс, 2019. 477 с.

8. Prokop Y. V., Bukata L. N., Trofymenko O. G. Algorithmization and Programming. Structured Data Programming. Odesa : A. S. Popov ONAT, 2020. 57 p.

9. ISO/IEC/IEEE 12207:2017. Systems and software engineering — Software life cycle processes. Geneva : ISO, 2017.

Інформаційні ресурси

10. ISO. Software life cycle processes (ISO/IEC/IEEE 12207:2017). URL: https://www.iso.org/standard/63712.html

11. Microsoft Learn. C++ documentation. URL: https://learn.microsoft.com/cpp

12. cppreference.com. C++ language reference. URL: https://en.cppreference.com

13. MIT OpenCourseWare. Introduction to Programming. URL: https://ocw.mit.edu

14. Algorithms Specialization (Coursera). URL: https://www.coursera.org/specializations/algorithms

Навчальне видання

Русу Олександр Петрович

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

Конспект лекцій для здобувачів вищої освіти,
які навчаються за спеціальностями галузі «Інформаційні технології»

Українською мовою