Алгоритмізація та програмування процедур обробки інформації

C++

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ

О. І. Щедріна

Алгоритмізація та програмування процедур обробки інформації

Навчальний посібник

Рекомендовано Міністерством освіти і науки України

Київ 2001

ББК 32.97
Щ 36

Рецензенти:

Г. В. Лавінський, д-р техн. наук, проф. (АКБ «ТК Кредит»)
Н. Р. Головко, канд. екон. наук, доц. (Київ. нац. екон. ун-т)

Гриф надано Міністерством освіти і науки України
Лист № 2/1170 від 11.07.2000

Щедріна О. І.

Щ 36 Алгоритмізація та програмування процедур обробки інформації: Навч. посібник. — К.: КНЕУ, 2001. — 240 с.
ISBN 966–574–226–4

У навчальному посібнику розглянуті основи алгоритмізації обчислювальних процесів і алгоритмізації процедур обробки соціально-економічної інформації, питання проектування та програмування прикладних програм, прийоми програмування на мові С++, синтаксис і семантика мови Borland С++. У посібнику наведена велика кількість прикладів програм, що ілюструють особливості мови С++, аналізуються типові помилки використання конструкцій мови.

Для студентів вищих навчальних закладів, що навчаються за спеціальністю «Економічна кібернетика», студентів та програмістів, які самостійно опановують мову С++.

ББК 32.97

Навчальне видання

ЩЕДРІНА Олена Іванівна

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ ПРОЦЕДУР ОБРОБКИ ІНФОРМАЦІЇ

Навчальний посібник

Редактор *Ю. Пригорницький.* Художник обкладинки *О. Стеценко*
Технічний редактор *Т. Піхота.* Коректор *І. Савлук*
Комп'ютерна верстка *Т. Мальчевської, І. Пантюхової*

Підписано до друку 14.03.01. Формат 60×84/16. Папір офсет. № 1.
Гарнітура Тип Таймс. Друк офсетний. Умов. друк. арк. 13,95.
Умов. фарбовідб. 14,17. Обл.-вид. арк. 15,88. Наклад 2000 прим. Зам. № 20-2015.

Видавництво КНЕУ
03680, м. Київ, проспект Перемоги, 54/1
Свідоцтво про реєстрацію №235 від 07.11.2000
Тел. (044) 458–00–66; тел./факс (044) 446–64–58
E-mail: publish@kneu.kiev.ua

Віддруковано в друкарні МВС України
вул. Дегтярівська, 15б.

ISBN 966–574–226–4

Передмова

Поява ПЕОМ набагато розширила коло користувачів обчислювальної техніки, змінився і склад програмного забезпечення ЕОМ, у якому більше місце займають готові прикладні системи, що полегшують користувачеві застосування ЕОМ для різних видів робіт. Тим не менше, постійно виникає потреба розширити та модифікувати функції готової системи. Саму прикладну систему також необхідно програмувати якою-небудь мовою високого рівня. Такою мовою програмування для ПЕОМ стала мова С++. Мова С++ вже стала універсальною мовою для програмістів цілого світу. Дана мова може бути застосована фактично для програмування будь-якої задачі. Мова С++ часто використовується для таких проектів, як створення операційних систем, редакторів, систем керування баз даних, мультимедіа-програм, персональних інформаційних систем для розв'язання економічних, наукових, інженерних задач. Більша частина сучасного системного та прикладного програмного забезпечення розробляється саме цією мовою.

Для вивчення системи програмування мовою С++ саме і призначена дисципліна «Алгоритмізація та програмування процедур обробки інформації» для підготовки бакалаврів з економіки, які навчаються за спеціальністю «Економічна кібернетика».

Метою вивчення дисципліни є набуття студентами знань та практичних навичок з алгоритмізації типових процедур обробки соціально-економічної інформації,

технології проектування та програмування, програмування власних програмних продуктів мовою C++.

Під час навчання слухачі дисципліни набувають практичних навичок з алгоритмізації та відпрацьовують типові елементи програмування, вивчають технологію програмування, набувають навичок налагодження та виконання програм у середовищі Borland C++, вивчають нові засоби програмування мовою C++, реалізують комплексні виробничі ситуації, пов'язані з проектуванням, програмуванням та документуванням програмних комплексів, організацією інтерфейсу з користувачем за допомогою створення вікон та меню, використання графічних можливостей системи.

Частина 1 ОСНОВИ АЛГОРИТМІЗАЦІЇ

Розділ 1

АЛГОРИТМ, СПОСОБИ ЙОГО ПОДАННЯ. ТИПИ АЛГОРИТМІЧНИХ ПРОЦЕСІВ ТА ПРИНЦИПИ ЇХ ПОБУДОВИ

1.1. ПОНЯТТЯ АЛГОРИТМУ

У своїй повсякденній діяльності нам постійно доводиться стикатися з різноманітними правилами, що приписують послідовність дій, мета яких складається в досягненні певного необхідного результату. Подібні правила дуже численні. Наприклад, ми дотримуємося певних правил, щоб зателефонувати, виконати певну послідовність дій, щоб зварити суп, приготувати ліки або обчислити добуток двох багатозначних чисел. Приклади такого роду називають алгоритмами.

Популярність поняття алгоритму в наш час зумовлена розвитком і широким застосуванням електронно-обчислювальної техніки. Використання ЕОМ сприяло тому, що розробка алгоритму — необхідний етап у процесі розв'язання задач на ЕОМ.

Поняття алгоритму, що належить до фундаментальних концепцій інформатики, виникло задовго до появи ЕОМ і стало одним з основних понять математики. Термін «алгоритм» походить від імені великого узбецького математика Мугаммада бен Муса аль Хорезмі (IX в.), algorithmi — це латинська транскрипція

імені аль-Хорезмі, це слово використовувалося в математиці для позначення правил виконання чотирьох арифметичних дій: додавання, віднімання, множення і ділення над числами в десятковій системі числення. Сукупність цих правил у Європі стали називати «алгоризм». Згодом це слово переродилося в алгоритм і зробилося збірною назвою окремих правил певного вигляду і не лише правил арифметичних дій. Протягом тривалого часу його вживали тільки математики, позначаючи правила розв'язання різних задач.

До аль-Хорезмі існувало дуже багато прийомів виконання арифметичних операцій, у яких враховувалися ті або інші особливості конкретних чисел, над якими проводилися ці операції. Наприклад, якщо один зі співмножників зображає число, що складається з одних дев'яток, то до іншого співмножника треба дописати число нулів, рівне числу дев'яток у першому співмножнику, і потім від добутого числа відняти другий співмножник. Наприклад, якщо число 999 потрібно помножити на 5, до цифри 5 слід додати три нулі, оскільки в першому співмножнику три дев'ятки, — отримаємо 5000, а потім відняти другий співмножник, тобто число 5. Результатом буде число 4995, яке і є добутком чисел 999 і 5. Ми досі користуємося багатьма прийомами швидкої усної лічби. Наприклад, щоб помножити будь-яке число на 5, необхідно дописати до нього 0 і розділити добуте число навпіл. Щоб помножити число на 25, досить помножити це число на 100 і розділити результат на 4. Однак такого роду прийоми можна застосовувати лише в тих випадках, коли хоч би один зі співмножників має спеціальний вигляд (всі дев'ятки, 5, 25 і т. д.). Аль-Хорезмі запропонував правила, придатні у всіх випадках і однакові для будь-яких пар чисел.

Алгоритм — це точне приписання, що визначає процес перетворення початкових даних для отримання кінцевих результатів під час розв'язання певного класу задач. У цьому точному приписанні задаються вказівки про виконання в певному порядку певної системи операцій і правила їх застосування до початкових даних для розв'язання задачі.

Поняття алгоритму — одне з основних понять математики, оскільки віднаходження припустимого алгоритму для різних класів задач є однією з цілей математики. Наведене визначення алгоритму не є точним математичним, а лише пояснює смисл даного слова. З появою обчислювальної техніки з'явилася необхідність уточнення поняття алгоритму як об'єкта математичної теорії. Це пояснюється тим, що виникли потреби в загальних способах формалізації і одноманітного розв'язання цілих класів задач на базі могутніх універсальних алгоритмів.

Спроби знайти та сформувати в суворих термінах прийоми й форми побудови алгоритмів, досить строгі в плані математичних досліджень і досить загальні, щоб виражати всілякі алгоритми, привели до виникнення самостійної дисципліни — теорії алгоритмів, у якій розкриваються теоретичні можливості розробки ефективних алгоритмів обчислювальних процесів і їх прикладних застосувань.

1.2. ВЛАСТИВОСТІ АЛГОРИТМУ (ALGORITHM PROPERTIES)

Під час розв'язання будь-якої задачі звичайно виходять з наявності деяких початкових даних і мають уявлення про результат, який треба одержати. Приступаючи до рішення, спочатку намічають план і послідовність дій, які від початкових даних повинні привести до шуканого результату, тобто прагнуть побудувати алгоритм розв'язання. Такий ланцюжок дій, кожна з яких, за припущенням, відомим способом здійснена, і складає алгоритм; його виконання називається *алгоритмічним процесом*, а кожна ланка цього ланцюжка — кроком алгоритмічного процесу, або алгоритму. Можна сказати, що алгоритм вказує послідовність дій з переробки початкових даних у результат.

Будь-який алгоритм має відповідати таким вимогам:

1. *Масовість* — застосовність алгоритму до будь-яких даних задач певного класу. Ця властивість алгоритму забезпечує розв'язання будь-якої задачі з класу однотипних задач при будь-яких початкових даних. Так, алгоритм обчислення площі трикутника застосуємо до будь-яких трикутників. Для алгоритму можна брати різні набори вхідних даних, тобто можна застосовувати один і той самий алгоритм для розв'язання цілого класу однотипних задач. Наприклад, алгоритм для розв'язання системи лінійних рівнянь повинен бути застосовний до системи, що складається з довільного числа рівнянь, причому початкові дані для алгоритму можна вибирати з певної множини. Разом з тим, існують і такі алгоритми, що застосовні лише до єдиного набору початкових даних. До їх числа належать алгоритми користування різними автоматами, наприклад, автоматом, що продає газети, або телефоном-автоматом, якщо вони розраховані на одну певну монету чи жетон, алгоритми слідування по маршруту, що починаються в певному пункті і ведуть в задане місце, й багато інших.

Яке ж значення потрібно вкладати в термін «масовість»? Слід вважати, що для кожного алгоритму існує певний клас об'єктів, і всі вони допустимі як початкові дані. Для алгоритмів виконання арифметичних операцій — додавання, віднімання, множення і ділення, такий клас складає всі дійсні числа. Масовість алгоритму — це допустимість усіх об'єктів відповідного класу, а зовсім не допустимість якоїсь їх кількості (кінцевого чи нескінченного). А скінченна або нескінченна кількість допустимих об'єктів — це вже властивість вказаного класу.

2. *Визначеність (детермінованість)(determinancy)* — набір вказівок має бути точний, не залежати від виконавця. Ця характеристика забезпечує визначеність, однозначність результату процесу, що описується ним при заданих початкових даних. Кожен крок повинен бути чітко та недвозначно визначений і не повинен допускати довільного трактування виконавцем. Під час виконання алгоритму виконавець має діяти суворо відповідно до його правил і в нього не має виникати потреби робити які-небудь дії, відмінні від продиктованих алгоритмом. Строго визначеним повинен бути і порядок виконання дій. Ця дуже важлива властивість означає, що якщо один і той самий алгоритм доручити для виконання різним виконавцям, то вони прийдуть до одного й того самого результату.

Наведемо приклад алгоритму поділу відрізка навпіл за допомогою циркуля й лінійки. Для того, щоб поділити відрізок навпіл, необхідно відкрити циркуль на довжину відрізка і з кінців відрізка описати цим радіусом коло. Через точки перетинів кіл проведемо пряму. Ця пряма перетне заданий відрізок у точці, яка і є серединою даного відрізка.

Аналіз алгоритмів показує, що коли застосовувати алгоритми повторно до одних і тих самих початкових даних, то діставатимемо один і той же результат. Наприклад, якщо застосовувати наведений вище алгоритм поділу відрізка навпіл до одного й того самого відрізка, то щоразу отримуватимемо одну й ту саму точку.

І якщо при цьому кожен раз порівнювати результати, отримані після відповідних кроків алгоритмічного процесу, то виявиться, що при одних і тих же початкових даних ці результати завжди будуть однаковими. Таким чином, можна говорити про визначеність і однозначність алгоритмів.

3. *Дискретність* — розчленованість процесу, що визначається алгоритмом, на окремі елементарні операції, можливість виконання яких людиною або машиною не викликає сумнівів. Процес, який визначається алгоритмом, повинен мати дискретний

характер, тобто являти собою послідовність окремих кроків. Виконання алгоритму розчленовується на виконання окремих кроків, виконання кожного чергового кроку починається після завершення попереднього.

4. *Зрозумілість* — знання виконавця про те, що треба робити для виконання цього алгоритму. При цьому виконавець алгоритму, виконуючи його, діє «механічно», тому формулювання алгоритму має бути настільки точне й однозначне, щоб могло повністю визначати всі дії виконавця. Людина, навіть не втаємничена в тій сфері, для розв'язання задач якої розроблений алгоритм, але яка розуміє вказівки, що містяться в ньому і точно їх виконує, — обов'язково дістає шукане розв'язання задачі.

5. *Результативність* — кінцівка процесу перетворення вхідної інформації у вихідну. Результативність вказує на те, що застосування алгоритму до будь-якого допустимого набору вхідних даних за кінцеве число кроків забезпечує отримання певного результату. Під час виконання алгоритму деякі його кроки можуть повторюватися багато разів, однак виконання алгоритму все ж закінчиться за кінцеве число кроків. Тобто смисл результативності полягає в тому, що алгоритмічний процес повинен бути кінцевим: за певне число кроків повинен бути добутий шуканий результат або доведено, що алгоритм не застосований до даної множини початкових даних. Непридатність алгоритму до допустимих початкових даних має місце або під час порушення умови скінченності алгоритмічного процесу, необхідною виявляється нескінченна кількість кроків алгоритму, або під час виникнення непереборних перешкод до його виконання на якомусь кроці.

У математиці є обчислювальні процеси, що мають алгоритмічний характер, але не володіють властивістю кінцівки. Так можна сформулювати процедуру обчислення числа π. Така процедура описує нескінченний процес і ніколи не завершиться. Якщо спосіб отримання наступної величини з якоїсь заданої не приводить до результату, то повинна бути вказівка, що треба вважати результатом алгоритму. У нашому випадку можна ввести умову завершення процесу обчислення виду: «Закінчити обчислення після отримання n десяткових знаків числа π», — то вийде алгоритм обчислення n десяткових знаків числа π. На цьому принципі базується отримання багатьох обчислювальних алгоритмів: будується нескінченний, збіжний до шуканого розв'язку процес. Він обривається на певному кроці, і добуте значення приймається за наближений розв'язок задачі, що розглядається. При цьому точність наближення залежить від числа кроків.

6. *Формальність* — результат виконання алгоритму не повинен залежати від будь-яких факторів, які не є частиною цього алгоритму. Будь-який виконавець, здатний сприймати і виконувати вказівки алгоритму (навіть не розуміючи їх змісту), діючи за алгоритмом, може виконати поставлене завдання. Ця властивість має особливе значення для виконання алгоритмів. Очевидно, електронно-обчислювальні машини не можуть розуміти суті завдань і окремих вказівок алгоритмів, хоча успішно виконують різні алгоритми.

Наведемо точніше визначення алгоритму. *Алгоритм — це скінченна послідовність однозначних розпоряджень, виконання яких дозволяє за допомогою скінченного числа кроків отримати розв'язання задачі, що однозначно визначається початковими даними.* Характеризує алгоритм його зрозумілість для виконавця; детермінованість (визначеність), тобто однозначність результату при заданих початкових даних; дискретність процесу, що визначається алгоритмом, — можливість розчленувати його на окремі елементарні операції, які легко здійснити; масовість — початкові дані, для яких застосуємо алгоритм, можна вибирати з певної множини допустимих значень даних, ця множина може бути як скінченною, так і нескінченною, тобто алгоритм повинен забезпечувати розв'язання будь-якої задачі з певного класу однотипних задач.

Незважаючи на зроблені уточнення поняття алгоритму, ми отримали уявлення про алгоритм в інтуїтивному значенні, яким і будемо надалі керуватися.

Подані вище коментарі пояснюють інтуїтивне поняття алгоритму, але саме це поняття не стає від цього більш чітким і строгим. Проте математики тривалий час задовольнялися цим поняттям. Лише з виявленням алгоритмічно нерозв'язних задач, тобто задач, для рішення яких неможливо побудувати алгоритм, з'явилася потреба в побудові формального визначення алгоритму, відповідного відомому інтуїтивному поняттю.

Інтуїтивне поняття алгоритму внаслідок своєї розпливчастості не може бути об'єктом математичного вивчення, тому для доказу існування алгоритму розв'язання задачі було необхідне строге формальне визначення алгоритму.

Побудова такого формального визначення була почата з формалізації об'єктів (операндів) алгоритму, оскільки в інтуїтивному понятті алгоритму його об'єкти можуть мати довільну природу. Ними можуть бути, наприклад, числа, дані датчиків, що фіксують параметри виробничого процесу, шахові фігури та позиції і т. ін. Однак, припускаючи, що алгоритм має справу не з самими ре-

альними об'єктами, а з їхніми зображеннями, можна вважати, що операнди алгоритму є слова в довільному алфавіті. Тоді виходить, що алгоритм перетворює слова в довільному алфавіті на слова того самого алфавіту. Подальша формалізація поняття алгоритму пов'язана з формалізацією дій над операндами і порядком цих дій. Одна з таких формалізацій була запропонована в 1936 р. англійським математиком А. Тьюрінгом, який формально описав конструкцію певної абстрактної машини, відомої під назвою машини Тьюрінга як виконавця алгоритму і висловив тезу про те, що всякий алгоритм може бути реалізований відповідною машиною Тьюрінга. Приблизно в цей же час американським математиком Е. Постом була запропонована інша формальна алгоритмічна схема — машина Поста, а 1954 року радянським математиком А. А. Марковим була розроблена теорія класу алгоритмів, названих ним нормальними алгорифмами, і висловлена основна теза про те, що всякий алгоритм може бути нормальним.

Ці алгоритмічні схеми є еквівалентними в тому значенні, що алгоритми, які описуються в одній зі схем, можуть бути також описані і в іншій, ці алгоритми об'єднуються під назвою логічні.

Логічні алгоритми цілком придатні для розв'язання теоретичних питань про існування чи неіснування алгоритму, але вони ніяк не допомагають у випадках, коли потрібно одержати добрий алгоритм, придатний для практичного застосування. З погляду логічних теорій, алгоритми, призначені для практичних застосувань, є алгоритмами в інтуїтивному значенні. Тому під час розв'язання проблем, що виникають під час створення й аналізу таких алгоритмів, нерідко керуються лише інтуїцією, а не строгою математичною теорією.

Таким чином, практика поставила задачу створення змістовної теорії, предметом якої були б алгоритми як такі і яка дозволяла б оцінювати їх якість, давала б практично придатні методи їх побудови, еквівалентного перетворення, доказу правильності і т. ін.

Змістовна (аналітична) теорія алгоритмів стала можливою лише завдяки фундаментальним роботам математиків у галузі логічних теорій алгоритмів. Розвиток такої теорії пов'язаний з подальшим розвитком і розширенням формального поняття алгоритму, яке дуже обмежене в рамках логічних теорій. Формальний характер поняття дозволить застосовувати до нього математичні методи дослідження, а його широта має забезпечити можливість охоплення всіх типів алгоритмів, з якими доводиться мати справу на практиці.

Докладний розгляд питань теорії алгоритмів виходить за рамки даного навчального посібника.

1.3. СПОСОБИ ПРЕДСТАВЛЕННЯ АЛГОРИТМІВ

Щоб довести до користувача алгоритми в залежності від їх призначення, вони мають бути формалізовані за певними правилами за допомогою конкретних зображальних засобів. Засоби, що використовуються для запису алгоритмів, значною мірою визначаються тим, для якого виконавця призначається алгоритм. Якщо алгоритм призначений для виконавця-людини, то його запис може бути не повністю формалізований, у цьому разі головне в формі запису — це наочність і зрозумілість. Для запису алгоритмів, призначених для реалізації на ЕОМ, необхідна строга формалізація. До основних зображальних засобів алгоритмів належать такі способи їх запису: словесний, формульно-словесний, схеми алгоритмів, мова операторних схем, HIPO-схеми, псевдокоди, мови програмування. Розглянемо основні, традиційні способи представлення алгоритму.

Словесний запис алгоритму. При даному способі запису алгоритму кожна операція перетворення формулюється природною мовою у вигляді правила. Правила нумеруються, щоб мати можливість на них посилатися, і зазначається порядок їх виконання. Ось приклад словесної форми запису алгоритму для знаходження найбільшого з трьох чисел. Маємо числа *a*, *b*, *d*. Знайти число *x*, рівне найбільшому з них. Спочатку знайдемо найбільше з двох чисел, наприклад, *a* і *b*. Якщо *a* менше, ніж *b*, то, припустімо, *x=b*, якщо ж *a* більше, ніж *b*, то, припустімо, *x=a*. Таким чином, внаслідок порівняння *a* з *b* ми отримуємо величину *x*, яка рівна найбільшому з них. Порівняємо тепер *x* і *d*. Якщо $x \geq d$, то значення *x* потрібно прийняти як результат. Якщо ж $x < d$, то *x* потрібно покласти рівним *d*. Таким чином, як результат добуваємо величину *x*, яка рівна найбільшому з *a, b, d*.

Алгоритм розв'язання даної задачі можна представити чіткіше таким чином:

1. Якщо $a \geq b$, то перейти до п. 4.
2. *x* покласти рівним *b*.
3. Перейти до п. 5.
4. *x* покласти рівним *a*.
5. Якщо $x \geq d$, то перейти до п. 7.
6. *x* покласти рівним *d*.
7. Виведення *x*.

Недоліком словесного способу представлення алгоритму є відсутність строгої формалізації і наочності, але ним можна описувати алгоритми з довільною мірою деталізування.

Формульно-словесний спосіб запису. Цей спосіб запису алгоритму ґрунтується на завданні інструкцій про виконання конкретних дій у певній послідовності з використанням математичних символів і виразів зі словесними поясненнями.

Наприклад, потрібно обчислити значення многочлена $P_n(x)$ степеня *n* в певній точці *x*. Многочлен можна зобразити в традиційній формі:

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0,$$

або у відповідності зі схемою Горнера:

$$P_n(x) = (\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0.$$

Наприклад, для n = 4

$$P_4(x) = (((a_4 x + a_3)x + a_2)x + a_1)x + a_0.$$

Словесно-формульний запис обчислення значення многочлена степеня *n* за схемою Горнера можливий, наприклад, такий:

1. i: = n.
2. S: = 0.
3. S: = S*x + a_i.
4. i: = i — 1.
5. Якщо $i \geq 0$, то перейти до п. 3.

Пункти 1, 2 виконуються по одному разу, а пп. 3—5 виконуються n + 1 разів.

Формульно-словесний спосіб запису алгоритму більш компактний і наочний в порівнянні зі словесним, але не є строго формалізованим. Даний спосіб прийнятий під час опису різного роду математичних викладок, доказів теорем. Він легко читається і зрозумілий багатьом фахівцям без спеціальної підготовки.

Іншим інструментом, що часто використовується для запису алгоритмів, є **схеми алгоритмів**. Схема алгоритму [flowchart, flow diagram] являє собою графічне зображення процесу розв'язання задачі у вигляді послідовності блоків спеціального вигляду, що відображають специфіку перетворення інформації і сполучених між собою лініями чи стрілками, що вказують черговість виконання блоків. Або інакше схема алгоритму програми — метод розробки програм, який використовує набір стандартних графічних зображень для представлення обчислювальних дій. Всередині кожного блоку стисло записується зміст конкретної ділянки розв'язання алгоритму задачі. Існує державний стандарт на умовні позначення (символи) в схемах алгоритмів, програм, даних і систем, він встановлює правила виконання схем, що ви-

користовуються для відображення різних видів задач обробки даних і засобів їх розв'язання. Оформлення схем алгоритмів і програм повинно виконуватися відповідно до вимог цього стандарту: «Єдина система програмної документації. Схеми алгоритмів, програм, даних і систем. Умовні позначення і правила виконання» ГОСТ 19.701-90 (ИСО 5807—85).

Схеми програм відображають послідовність операцій у програмі. Схема програми складається з:

1) символів процесу, що вказують фактичні операції обробки даних (включаючи символи, що визначають шлях, якого потрібно дотримуватися з урахуванням логічних умов);

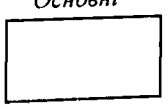
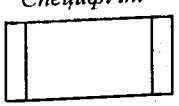
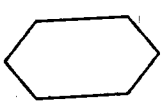
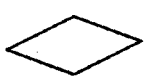
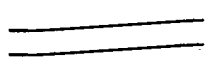
2) лінійних символів, які вказують потік керування;

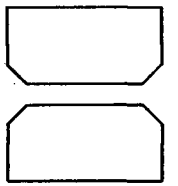
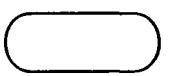
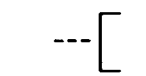
3) спеціальних символів, що використовуються для полегшення написання і читання схеми.

Прийняті графічні зображення блоків наведені в табл. 1.1.

Таблиця 1.1

ОСНОВНІ СИМВОЛИ СХЕМ АЛГОРИТМІВ. ЗАСТОСУВАННЯ СИМВОЛІВ

Символ	Найменування символу	Функція
Символи процесу		
Основні		
	Процес	Виконання певної операції або групи операцій, що призводить до зміни значення, форми або розміщення інформації чи до визначення, за яким з декількох напрямків потоку потрібно рухатися.
Специфічні		
	Зумовлений процес	Відображає процес, що складається з однієї або кількох операцій, або кроків програми, які визначені в іншому місці (в підпрограмі, модулі).
	Підготовка	Відображає модифікацію команди чи групи команд з метою впливу на певну наступну функцію (установка перемикача, модифікація індексного регістра або ініціалізація програми).
	Розв'язання	Вказує вибір напряму виконання алгоритму чи програми в залежності від певних змінних умов.
	Паралельні дії	Відображає синхронізацію двох або більше паралельних операцій.

Символ	Найменування символу	Функція
	Межа циклу	Символ складається з двох частин і відображає початок і кінець циклу. Обидві частини символу мають один і той самий ідентифікатор. Умови для ініціалізації, приросту завершення і т. д. вміщуються всередині символу на початку чи в кінці в залежності від розташування операції, що перевіряє умову.
Символи ліній		
Основні		
	Лінія	Символ відображає потік даних або керування. При необхідності або для підвищення зручності читання можуть бути додані стрілки-покажчики.
Специфічні		
	Пунктирна лінія	Символ відображає альтернативний зв'язок між двома або більше символами, а також використовується для обведення анотованої ділянки.
Спеціальні символи		
	З'єднувач	Відображає вихід у частину схеми і вхід з іншої частини цієї схеми і використовується для обриву лінії і продовження її в іншому місці. Відповідні символи-з'єднувачі повинні містити одне й те саме унікальне позначення.
	Термінатор	Визначає початок і кінець схеми програми, зовнішнє використання і джерело або пункт призначення даних.
	Коментар	Використовують для додавання описових коментарів або пояснювальних записів з метою пояснення чи приміток.
	Пропуск	Використовують у схемах для відображення пропуску символу або групи символів, у яких не визначені ні тип, ні число символів.

У схемах алгоритмів для обчислення значення змінних звичайно користуються знаком : = (присвоїти). Це дозволяє відрізняти запис y:=a, який читається як «змінній y присвоїти значення, рівне a», від y=a, що означає перевірку рівності двох змінних.

Розрізняють укрупнену та детальну схеми алгоритму. Укрупнена схема алгоритму зображає обчислювальний процес в укрупненому плані на рівні типових процесів обробки даних, що залежать від класу задач, що вирішуються. Наприклад, під час обробки економічних даних типовими процесами є введення даних, коректування, сортування, обробка даних і т. ін. До складання детальних схем алгоритмів приступають після ретельного аналізу укрупнених схем алгоритмів. Кожен блок укрупненої схеми алгоритму представляється у вигляді окремої схеми, що деталізує схему обчислення. У детальній схемі алгоритму враховуються в деталях усілякі логічні зв'язки між етапами переробки даних й елементарні операції, що виконуються в них.

На рис. 1.1 представлений у вигляді схеми алгоритм визначення розв'язку квадратного рівняння.

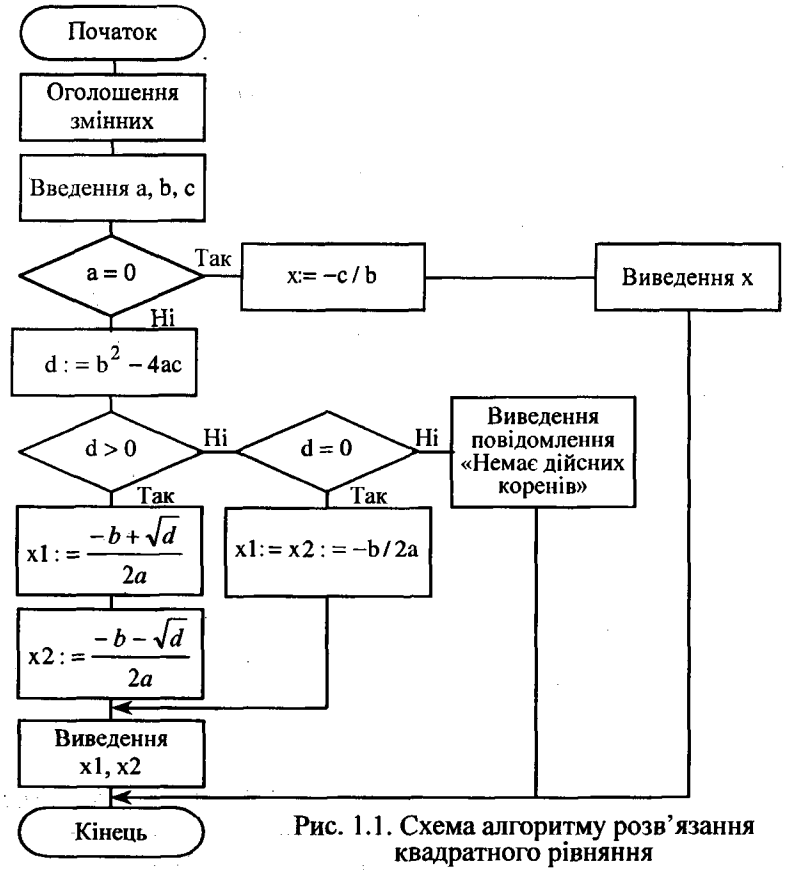


Рис. 1.1. Схема алгоритму розв'язання квадратного рівняння

Схеми алгоритмів мають більшу наочність, ніж словесний запис алгоритму. Схема дозволяє чітко уявити собі логічні зв'язки між частинами алгоритму, простежити за послідовністю дій в алгоритмі, перевірити, чи всі можливі варіанти задачі знайшли в ньому відображення. Проте ця наочність швидко втрачається під час зображення скільки-небудь великого алгоритму — в цьому разі схема виходить погано доступною для огляду.

Наступним зображальним засобом алгоритму є операторні схеми. Операторний метод запису алгоритму був розроблений радянським математиком О. А. Ляпуновим у 1953 р. Операторна схема — аналітична форма представлення алгоритму за допомогою операторів, що діють на деякі елементи інформації. Для запису алгоритмів застосовують такі основні типи операторів: арифметичні, логічні, оператори переадресації і відновлення, оператори перенесення, оператори формування. Операторна схема показує, з яких операторів перелічених вище типів складатиметься алгоритм розв'язання задачі і яким є відносне розташування цих операторів.

Операторна форма запису алгоритмів використовує для опису їх структури спеціальні символи, кожен з яких відповідає певному оператору. Вибір букв для позначення операторів довільний.

Арифметичні оператори служать для запису різних арифметичних дій і позначаються початковими буквами латинського алфавіту.

До операторів перевірки логічних умов вдаються для визначення порядку роботи алгоритму і позначають їх малими буквами латинського алфавіту.

Оператори переадресації і відновлення служать для зміни різних параметрів і адрес, від яких залежать оператори програми; для відновлення значень параметрів і адрес, які були змінені в процесі роботи алгоритму. Оператори переадресації і відновлення позначають літерою F з вказівкою в дужках змінної адреси чи параметра. Наприклад, оператор, що змінює і на одиницю, буде означатися F(i).

Оператор перенесення використовують для перенесення одного параметра на місце іншого, тобто для заміни одного параметра іншим. Оператори перенесення позначають: [a → b].

Оператор формування призначений для формування початкових значень деяких операторів. Вони переносять деякі заздалегідь занесені накази в певні місця алгоритму. Наприклад, { l → i } означає, що початкове значення параметра і має дорівнювати l.

В операторній схемі алгоритм зображається у вигляді послідовності операторів, що записуються в один рядок. Оператори забезпечуються номерами-індексами відповідно до порядку їх розташування в схемі. При цьому керуються такими правилами:

1) оператори мають наскрізну порядкову нумерацію незалежно від їх призначення;

2) якщо символи двох операторів стоять поруч, то це означає, що оператор, який стоїть праворуч, отримує керування від сусіднього зліва;

3) якщо оператор, що стоїть праворуч, не отримує керування від оператора, що стоїть ліворуч, то між цими операторами ставиться крапка з комою;

4) передача керування до оператора, що не стоїть праворуч поряд, позначається стрілкою.

Виконання алгоритму починається з найлівішого оператора схеми. Після цього виконується оператор, записаний праворуч від нього. Якщо це логічний оператор, то перевіряється умова задана ним. Під час виконання цієї умови черговим стає оператор, що стоїть праворуч від нього. В іншому разі, коли логічна умова не дотримується, оператор, що підлягає черговому виконанню, вказується стрілкою, що бере початок від даного логічного оператора до цього чергового оператора. Для компактності записів на початку і кінці стрілок можна ставити номери, знак $\uparrow_i$ означає початок стрілки, а знак $\downarrow_i$ — її кінець. Однаковими номерами позначаються початок і кінець однієї і тієї самої стрілки. Виконання алгоритму закінчується тоді, коли останній оператор містить вказівку про його припинення або коли на певному етапі не виявляється такого елемента схеми, який повинен був би виконуватися.

За допомогою операторної схеми подамо алгоритм знаходження найбільшого з трьох нерівних чисел. У табл. 1.2 наведені символи-оператори, що використовуються в операторній схемі алгоритму даного прикладу.

Таблиця 1.2

СИМВОЛИ-ОПЕРАТОРИ ОПЕРАТОРНОЇ СХЕМИ

№ п/п	Символ-оператор	Етап обробки даних, що описується символом-оператором
1	B_1	Введення початкових даних x, y, z
2	$p_2(x > y)$	Перевірка виконання логічної умови x > y
3	$[x \to R]$	R присвоїти значення x
4	$[y \to R]$	R присвоїти значення y
5	$p_5(z > R)$	Перевірка виконання логічної умови z > R
6	$[z \to R]$	R присвоїти значення z
7	$П_7$	Друк результату обчислення R
8	O_8	Зупинка

Операторну схему алгоритму можна представити так:

$$B_1 p_2(x > y)[x \to R];\ [y \to R]\, p_5(z > R)[z \to R]\, П_7 O_8 .$$

Метод операторних схем дав можливість більш формалізовано записувати алгоритм. Ця форма запису дозволяє зобразити алгоритм найкомпактніше, але не розкриває змісту операторів, тому в складних схемах потрібна розшифровка їхніх значень, тобто залучення текстуальної форми. Цей метод не знайшов широкого поширення. В рамках операторного методу був побудований ряд формальних мов, що дозволяють еквівалентно перетворювати схеми алгоритмів.

Псевдокод (Pseudecode). Псевдокод — мова, що нагадує мову програмування, але використовується для опису програми в загальних рисах, зображає один з методів складання програм. Псевдокод являє собою систему позначень і правил, призначену для одноманітного запису алгоритмів. Він займає проміжне місце між природною і формальною мовами.

Псевдокод близький до звичайної природної мови, тому алгоритми можуть на ньому записуватися і читатися як звичайний текст. З іншого боку, в псевдокоді використовуються деякі формальні конструкції і математична символіка, що наближає запис алгоритму до загальноприйнятого математичного запису.

У псевдокоді не прийняті строгі синтаксичні правила для запису команд, властиві формальним мовам, що полегшує запис алгоритму на стадії проектування і дає можливість використати ширший набір команд, розрахований на абстрактного користувача. Проте в псевдокоді є деякі конструкції, властиві формальним мовам, що полегшує перехід від запису на псевдокоді до запису алгоритму формальною мовою. У псевдокоді, так само як і в формальних мовах, є службові слова, значення яких визначене раз і назавжди. Вони виділяються в друкованому тексті жирним шрифтом, а в рукописному — підкреслюються.

Єдиного або формального визначення псевдокоду не існує, тому можливі різні псевдокоди, що відрізняються набором службових слів і основних конструкцій.

Як приклад приведемо запис псевдокоду обчислення факторіала, відомо, що $n! = 1 \cdot 2 \cdot 3 \ldots (n-1) \cdot n$.

алгоритм *факторіал;*
початок
    введення (n);
    f: = 1;

```
k: = 0;
поки k ≠ n
   повторювати
      початок
         k: = k+1;
         f: = f×k;
      кінець;
   виведення (f)
кінець
```

Елементарною структурною одиницею будь-якого алгоритму є проста команда, що позначає один елементарний крок переробки чи відображення інформації.

Запис алгоритму на псевдокоді починається із заголовка алгоритму, що містить службове слово **алгоритм**, за яким іде ім'я алгоритму (ідентифікатор). Ідентифікатор не може містити пробілу, тому, якщо ім'я складається з кількох слів, то ці слова пишуться разом, наприклад, **алгоритм** *сумадодатних*. Наявність імені в заголовку дозволяє надалі посилатися на алгоритм.

Змінній величині може бути присвоєне значення і за допомогою команди введення, яка передає виконавцеві значення змінної з певного зовнішнього джерела. Наприклад, команда

введення (n)

означає, що виконавець набуває із зовнішнього джерела значення, яке має бути присвоєне змінній n. Команда виведення

виведення (f)

означає, що виконавець повинен видати для огляду значення величини f.

З простих команд і перевірки умов утворюються складені команди, основні типи складених команд алгоритму є команда слідування, команда вибору, команда повторення.

Команда слідування утворюється з послідовності команд, наступних одна за іншою. На псевдокоді команди відділяються одна від іншої за допомогою крапок з комою. Під час виконання алгоритму команди виконуються одна за однією в тому порядку, як вони записані. Службові слова **початок** і **кінець** використовуються для позначення початку і кінця складених команд, ці службові слова виконують роль дужок. Наявність дужок дозволяє розглядати складений оператор як єдину дію, що розпадається на послідовність простіших дій.

Команда слідування може бути записана так:

початок <дія>; <дія>; ...; <дія> **кінець**

Під дією розуміємо або просту, або складену команду. Ці команди можуть записуватися або в рядок, або в стовпець — одна під іншою.

За допомогою команди вибору здійснюється вибір однієї з двох можливих дій у залежності від умови. На псевдокоді цю команду можна записати так:

якщо <умова>
   то <дія 1>
   інакше <дія 2>
все

Дії, вказані після службових слів **то** і **інакше**, можуть бути простими або складеними командами. Під час виконання команди вибору виконується тільки одна з дій: якщо умова істинна, то виконується дія 1, в іншому разі — дія 2.

Команда вибору може використовуватися в скороченому вигляді, в разі недотримання умови ніяка дія не виконується:

якщо <умова>
   то <дія 1>
все

Для позначення дій, що багато разів повторюються, використовують команду повторення, вона містить умову, що використовується для визначення кількості повторень. Команда повторення буває двох видів з передумовою і з післяумовою.

Команда повторення з передумовою записується ось у якому вигляді:

поки <умова>
   повторювати <дію>

Спочатку перевіряється умова. Якщо вона істинна, то виконується команда, записана після службового слова **повторювати**. Після цього знову перевіряється умова. Виконання циклу завершується, коли умова стане помилковою. Для цього необхідно, щоб команда, що виконується в циклі, впливала на умову.

Команда повторення з післяумовою виконується таким чином: умова перевіряється після виконання команди, а повторення виконання команди відбувається в тому разі, коли умова не дотримана, тобто повторення проводиться до перевірки умови. На псевдокоді команда повторення з післяумовою записується у вигляді

повторювати <дію>
до <умова>

Під час запису алгоритму на псевдокоді виходить текст, який можна читати без перерви згори донизу, як звичайний текст.

Внаслідок своїх особливостей псевдокоди, як і інші описані вище засоби запису алгоритмів, орієнтовані на людину.

Таблиця розв'язків [decision table]. Таблиця розв'язків — опис дій, які мають бути виконані під час різних комбінацій умов у вигляді матриці зі стовпцями, відповідними комбінаціям умов, і рядками, відповідними діям. Таблиця, в якій вказані всі можливі ситуації, що являють інтерес під час розгляду задачі, і дії, які повинні бути зроблені в залежності від різних умов, що виникають. У таблиці розв'язків у наочній формі представляється відповідність умов, що підлягають перевірці в певному процесі, і дій, що виконуються в разі задоволення умов. Таблиці розв'язків зручні для вираження алгоритмів, у яких є множина шляхів розв'язання задачі, причому шляхи визначаються комбінацією значень істинності заданої множини умов. Існують різні модифікації таблиць розв'язків, звичайно таблиця розв'язків представляється у вигляді певної таблиці, розділеної на дві таблиці: таблиці стану і таблиці рішень.

Наприклад, обчислити значення функції y, заданої формулою

$$y = \begin{cases} x^2 & \text{при } x < 0, \\ x + 1 & \text{при } x \geq 0. \end{cases}$$

Алгоритм обчислення цієї функції подамо в таблиці розв'язків.

Дія	Стан		
	x < 0	x > 0	x = 0
x^2	1	0	0
x + 1	0	1	1

У таблиці стану перераховуються прості умови, що підлягають перевірці. У таблиці дій задаються дії, що підлягають виконанню, а також зазначається, для яких комбінацій істинності умов дії виконуються чи ні. Як позначення для значень істинності звичайно використовують символи «ТАК» або «1», що означають, що відповідну дію потрібно виконати, «НІ» або 0 означає, що дія не виконується.

Таблиці розв'язків зручні для застосування при формулюванні логіки імітаційних моделей, у задачах управління виробництвом, для прийняття рішення, для формулювання запитів у базах даних та ін.

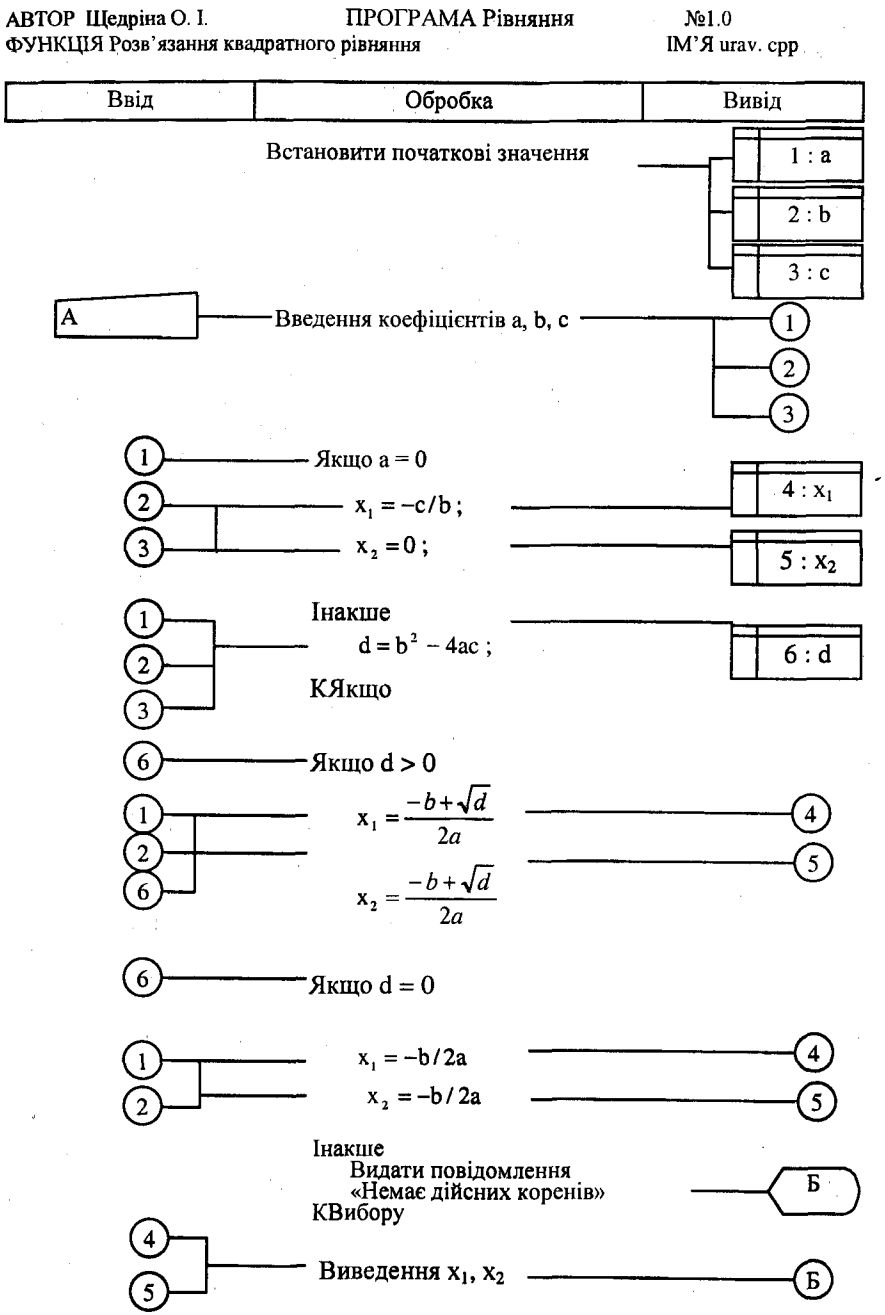
HIPO-схеми [Hierarchcal Input Process Output Diagramms]. HIPO-технологія — це багаторівнева дисципліна проектування та

документування програм. Технологія забезпечує поєднання на кожному етапі процесу розчленування певної функції алгоритму на ряд незалежних за керуванням підфункцій і планування обміну інформацією між ними. При цьому передбачається чіткий опис усіх функцій алгоритму від самого верхнього до нижнього рівнів. HIPO-технологія передбачає створення документації, що супроводжує процес розробки програм на всіх етапах. Ця документація забезпечує наочне зображення у формалізованому вигляді ієрархічної структури програми, всіх її функцій, вхідних і вихідних даних, що використовується і що генерується цими функціями, дозволяє простежити всі потоки обміну інформацією в програмі.

HIPO-схеми є основними в технології і кожна з них складається з чотирьох розділів: заголовка, входу, процесу (або обробки), виходу.

Заголовок схеми містить загальну інформацію про функцію, що описується: її коротке змістове найменування, ім'я відповідного програмного блоку, ідентифікаційний номер, прізвище автора і найменування блоку. Назва, що присвоюється програмному блоку на одній схемі, має бути короткою, загальнозрозумілою і обов'язково зберігатися без змін на всіх схемах розробки. При необхідності кожна назва може бути уточнена, розширена і пояснена в спеціальній області HIPO-схеми, яка називається областю специфікацій. У цій області можуть бути записані певні рекомендації з того, «як» реалізовується блок на даному рівні розгляду.

У першій колонці записується вхідна інформація, що на вході алгоритму, в останній — вихідна, що на виході, а в другій описаний процес, тобто алгоритм, який інформацію на вході перетворює у вихідну. Мова заповнення HIPO-схеми може бути будь-якою, в якій називається «що», а не «як» робить кожен програмний модуль на даному рівні проектування. Кожна HIPO-схема відповідає одному етапові (кроку) проектування. На одній схемі має бути не більше 6—7 програмних блоків, що уточнюють назву (характер роботи) даної схеми. Всі HIPO-схеми мають строго формалізовану систему посилань, якщо якийсь блок уточнюється іншими HIPO-схемами, то йому присвоюється відповідний номер. Якщо блок не має уточнюючого номера зв'язку, то його деталізація закінчена і він може бути безпосередньо закодований відповідною мовою програмування. Згідно з HIPO-технологією процес проектування системи закінчується лише після закінчення заповнення всіх HIPO-схем проекту і ув'язки їх одна з одною.



Дуже важливим в HIPO-технології є явне зазначення даних на вході і виході кожного алгоритму — те, що в звичайній технології утримується програмістом у думці під час побудови відповідної програми. У HIPO-технології дані на вході й виході вказуються явно, пов'язані з процесом обробки і вміщені структурно зліва і праворуч проектного аркуша. Цим HIPO-технологія істотно відрізняється від описаних вище способів запису алгоритмів.

Алгоритмічні мови. Нині найдовершенішим засобом для запису алгоритму є алгоритмічні мови, які дозволяють автоматизувати процес програмування за рахунок того, що переклад з алгоритмічних мов мовою машини здійснюється автоматично самою машиною за допомогою спеціальних програм-трансляторів.

Поява алгоритмічних мов зблизила поняття алгоритму і програми, оскільки з їх допомогою стало можливо записувати алгоритм розв'язання задачі, який одночасно служить початковою програмою, що безпосередньо сприймається машиною.

Для опису алгоритмів використовуються різні способи, що різняться мірою деталізування і формалізації. Теоретичний опис звичайно дається в словесному викладі, мета якого — обґрунтувати без зайвих подробиць процедуру, що пропонується як алгоритм. Для наочного представлення структури алгоритму широко використовуються схеми алгоритму. Формальний і повний опис алгоритмів здійснюється алгоритмічними мовами.

1.4. ТИПИ АЛГОРИТМІЧНИХ ПРОЦЕСІВ

Сукупність обчислювальних процесів, яка використовується для різного роду математичних, науково-технічних, економічних й інших задач, може бути розділена на три групи: лінійні, розгалужені і циклічні процеси.

Лінійним називається такий обчислювальний процес, у якому обробка інформації проводиться в суворій послідовності, що не залежить від значень даних, які обробляються. На схемі алгоритму лінійний обчислювальний процес подається послідовністю блоків. При цьому на схемі блоки, що виражають автономні етапи обчислень, розміщуються згори донизу в порядку їх виконання.

Наприклад, скласти схему алгоритму обчислення арифметичного виразу

$$y = \frac{x^2 + x + 1}{x^2 + 1}.$$

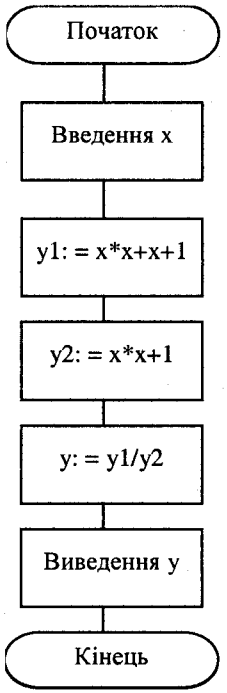


Рис. 1.2. Схема алгоритму лінійного процесу

Лінійні алгоритмічні процеси є основною частиною будь-якого алгоритму. Проте в практиці розв'язання задач цілковито лінійні задачі зустрічаються надто рідко. У багатьох випадках, у залежності від значень початкових даних або проміжних результатів, необхідно продовжити перетворення інформації за одним з кількох передбачених напрямів. Кожен окремий напрям обчислень називають віткою обчислень. Вибір тієї чи тієї вітки обчислень здійснюється перевіркою виконання логічної умови, що визначає властивості початкових даних і проміжних результатів. У кожному конкретному випадку процес реалізовується тільки за однією віткою, а виконання інших виключається. Подібні процеси обробки називаються розгалуженими.

Розгалуженим називається процес, у якому реалізація обчислень відбувається за одним з кількох заздалегідь передбачених напрямів у залежності від початкових значень або проміжних результатів. Розгалужені алгоритми бувають прості і складні. Простий процес, що розгалужується, — це такий процес, коли перша перевірка дає однозначний результат виконання. Складний — коли перша перевірка умови не дає однозначного результату, а вимагає перевірки наступних умов.

В основі організації розгалужень лежить перевірка значень ознаки та перехід на ту чи ту вітку обчислювального процесу в залежності від значення цієї ознаки.

На рис. 1.3 представлений алгоритм простого розгалуженого процесу, а на рис. 1.4 — складного.

У процесі розв'язання реальних задач досить часто спостерігається багаторазове повторення окремих етапів обчислювального процесу. Обчислювальні процеси, в яких одна або кілька операцій неодноразово повторюються, називаються **циклічними**. Повторення етапів обчислень у циклічному процесі відбувається щоразу з використанням нових значень змінних, які називаються

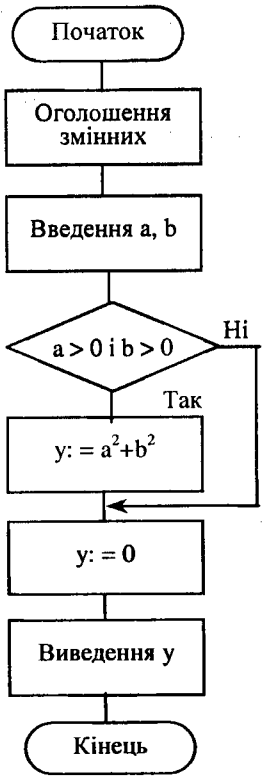


Рис. 1.3. Схема алгоритму простого розгалуженого процесу

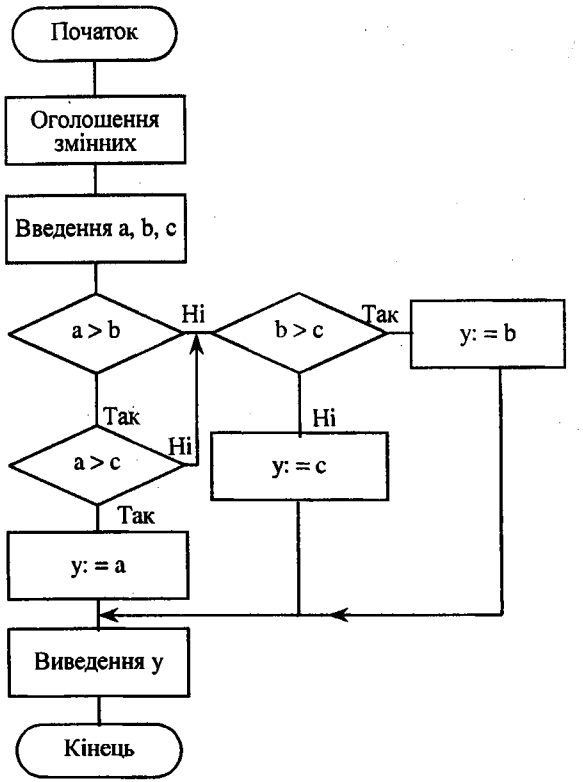


Рис. 1.4. Схема алгоритму складного розгалуженого процесу

параметрами циклу. Виконання циклічного обчислювального процесу зводиться до багаторазового обчислення за однією і тією самою математичною залежністю при різних значеннях величин, що входять до них. Цикли є основними частинами всіх програм, що практично використовуються, тому організація циклу є таким завданням програмування, що найчастіше зустрічається.

Якщо всередині циклу містяться інші цикли, то вони називаються складними, в іншому разі — простими. Ступінь вкладеності в даний цикл інших циклів називається рівнем вкладеності (nesting level). У багатьох мовах програмування така вкладеність лімітується звичайно до шести рівнів.

Як приклад розглянемо знаходження суми добутку елементів векторів $A = [a_1, a_2, \ldots, a_{10}]$ і $B = [b_1, b_2, \ldots, b_{10}]$ за формулою

$$S = a_1b_1 + a_2b_2 + \ldots + a_{10}b_{10} = a_ib_i. \qquad (1.1)$$

Для отримання значення S необхідно 10 разів виконати операцію множення елементів векторів і операцію додавання. Під час кожного виконання даних операцій до попереднього результату додається значення добутку чергових елементів векторів. Тобто в циклі виконується ділянка алгоритму вигляду

$$S: = S + a_ib_i.$$

Шляхом послідовної зміни індексу і від 1 до 10 з кроком 1 забезпечується перебір усіх елементів векторів для множення і додавання в циклі. Отже, циклом керує індекс змінної і, яка називається параметром циклу.

Схема алгоритму циклічного процесу наведена на рис. 1.5.

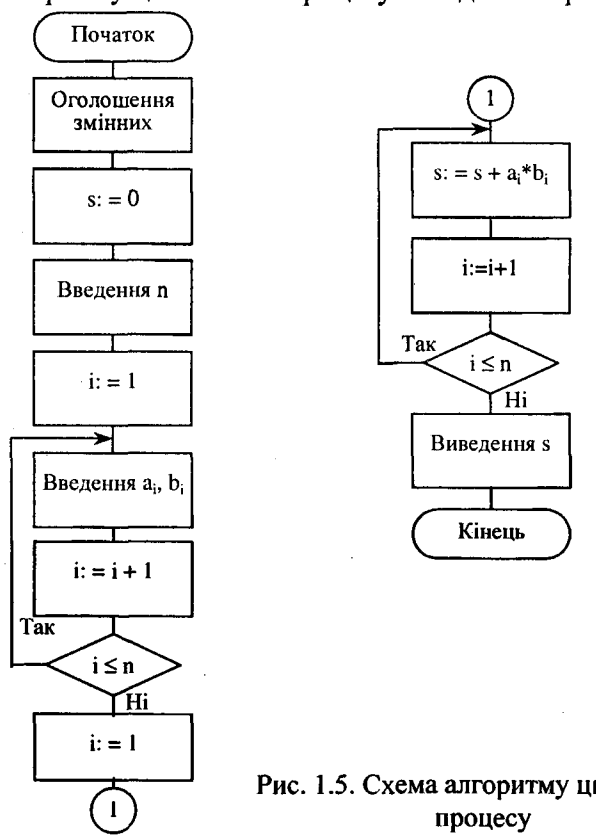


Рис. 1.5. Схема алгоритму циклічного процесу

У залежності від обмеження числа повторень циклу розрізняють цикли з відомою кількістю повторень та ітераційні цикли.

Для циклів з відомою кількістю повторень характерне завдання початкового і кінцевого значення параметра циклу, правило зміни параметра циклу під час кожного його повторення, кількість необхідних повторень циклу.

Наприклад, вищерозглянутий алгоритм обчислення виразу (1.1) реалізовується за допомогою циклу з відомою кількістю повторень. Початкове значення параметра циклу і дорівнює 1, кінцеве — 10. Початкове значення змінюється за правилом арифметичної прогресії з кроком 1. Таким чином, кількість необхідних повторень циклу становить 10. Оскільки в даний цикл не входять інші цикли, цикл є простим.

Процес обчислень, що ґрунтується на повторенні послідовності операцій, і на кожному кроці повторення використовується результат попереднього кроку, називається ітерацією. Тоді *ітераційним* називається обчислювальний процес, у якому для визначення подальшого значення змінної використовується її попереднє значення. Під час використання ітераційних процесів реалізовується метод послідовних наближень. Тож ітераційні циклічні процеси базуються на тому, що шуканій величині задається певне початкове значення, будується ітераційна формула-уточнення цього початкового значення до заданої точності. Ітераційні співвідношення будуються за принципом рекурсивних відношень, коли кожне нове значення віднаходиться через попереднє. Процес уточнення шуканої величини продовжується доти, доки різниця між заданими значеннями не стане меншою від заданої точності.

Для прикладу розглянемо алгоритм обчислення кореня p-го ступеня за ітераційною формулою:

$$y_{n+1} = \frac{1}{p}((p-1) \times y_n + \frac{x}{y_n^{p-1}}). \qquad (1.2)$$

Погрішність обчислень $|y_{n+1} - y_n| < \varepsilon$ початкове наближення $y_n = h$.

Шляхом послідовного наближення до заданого значення є за даною ітераційною формулою виходить шуканий результат кореня. Кількість повторень циклу до реалізації обчислювального процесу невідома. У цьому разі циклом керує задана погрішність обчислень ε. Якщо на черговій ітерації погрішність ≥ ε, то цикл продовжується для обчислення подальшого наближення значення результату y_{n+1}, інакше відбувається вихід з циклу.

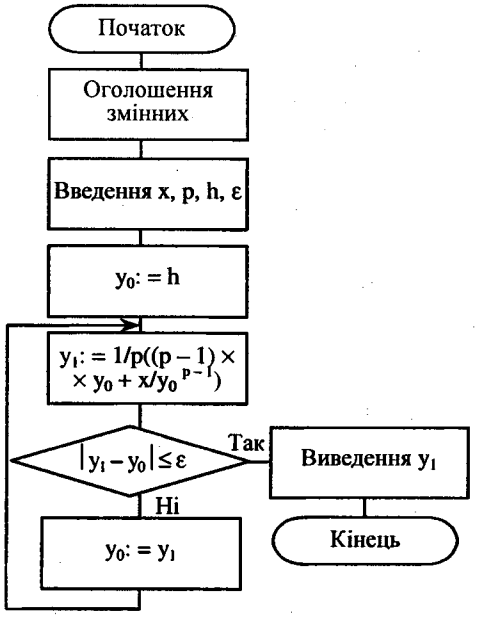


Рис. 1.6. Схема алгоритму обчислення кореня p-го ступеня

На рис. 1.6 наведена схема алгоритму для знаходження p-го ступеня величини x за формулою (1.2).

При значенні погрішності $|y_1 - y_0| \geq \varepsilon$ цикл продовжується. У блоці 6 результат попередньої ітерації y_0 заміниться результатом, отриманим при подальшій ітерації y_1. Після цього керування передається на блок 4, де за ітераційною формулою виходить чергове наближення значення кореня y_1 необхідної точності обчислень. Коли досягнута точність $|y_1 - y_0| < \varepsilon$, здійснюється вихід їх циклу шляхом передачі керування блоку 7.

З циклів з відомою кількістю повторень і ітераційних циклів можна будувати складні циклічні процеси з вкладеними циклами. Цикли, що включають інші цикли, заведено називати зовнішніми, а цикли, що входять у зовнішні цикли, — вкладеними. Для кожного циклу вибирається свій параметр і циклічний процес будується таким чином, що фіксується кожне значення параметра зовнішнього циклу і для нього параметр вкладеного циклу приймає всі свої значення. Якщо цикл є вкладеним, то він має закінчитися до того, як зовнішній цикл дійде кінця.

Розглянемо другий приклад ітераційного циклу. Потрібно обчислити наближеними методами значення ln(1 + x) за формулою

$$\ln(1+x) = \sum_{k=1}^{\infty} (-1)^{k+1} \frac{x^k}{k}$$

з точністю ε, яка вводиться з клавіатури. Вважати, що точність, яка потрібна, досягнена, якщо доданок не стане за модулем меншим від заданої точності. Значення функції обчислити для x від 0 до 1 з кроком 0,1.

Схема алгоритму наведена на рис. 1.7.

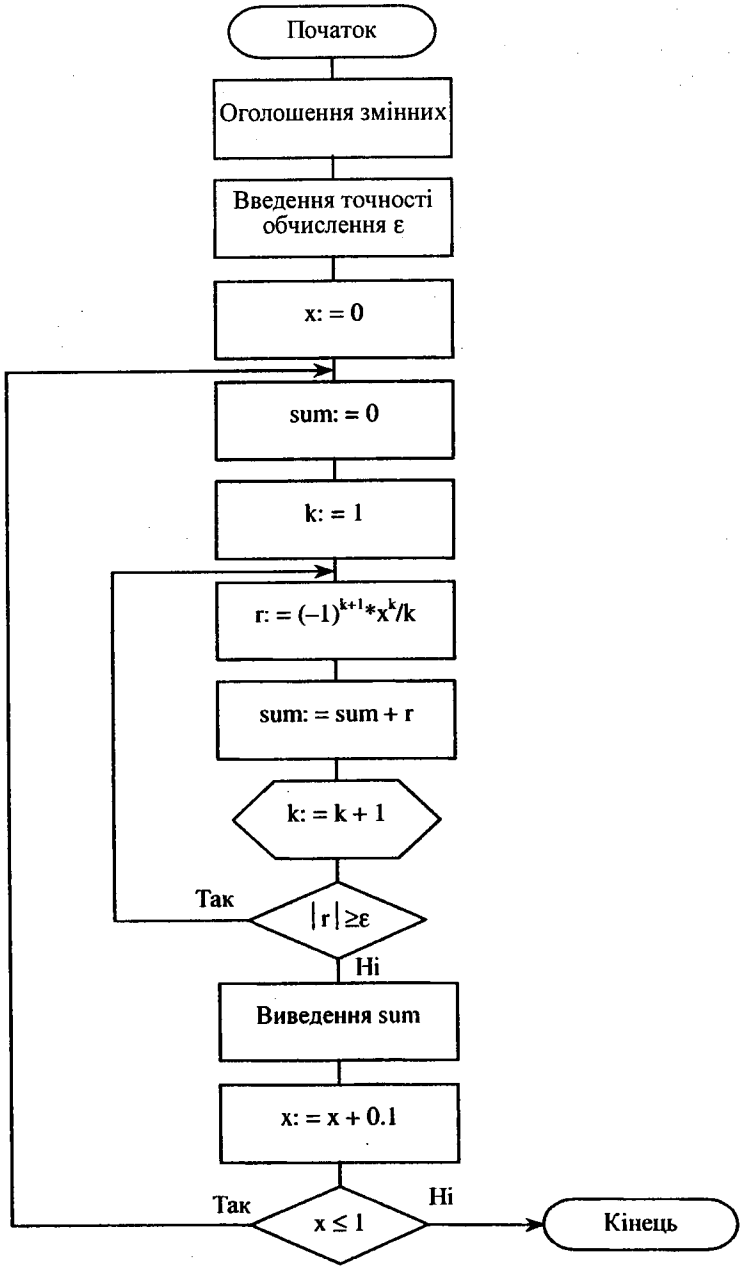


Рис. 1.7. Схема алгоритму складного ітераційного процесу

Складемо схему алгоритму для розрахунку суми і кількості додатних і від'ємних елементів матриці $A = [a_{ij}]_{m,n}$, де a_{ij} — елементи матриці; перший індекс i (i = 1, 2, ..., m) означає номер рядка, а другий елемент j — номер стовпця (j = 1, 2, ..., 3). Схема алгоритму наведена на рис. 1.8.

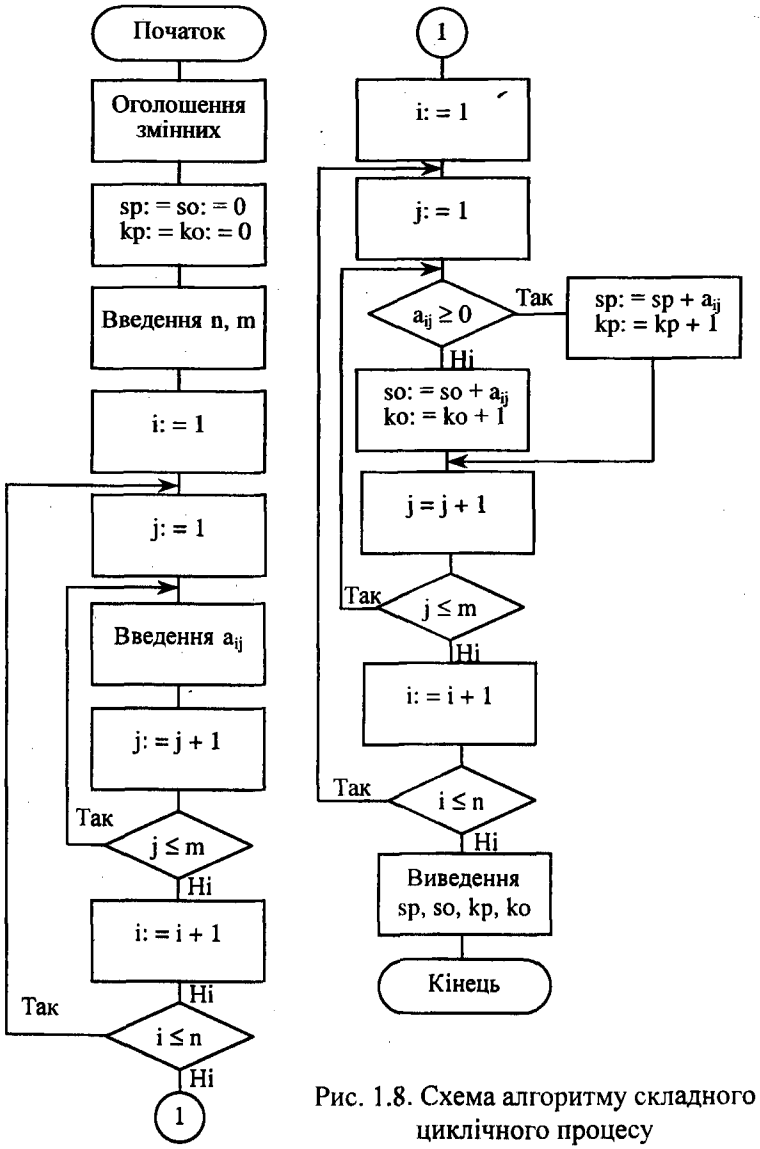


Рис. 1.8. Схема алгоритму складного циклічного процесу

Розділ 2

АЛГОРИТМІЗАЦІЯ ПРОЦЕДУР ОБРОБКИ СОЦІАЛЬНО-ЕКОНОМІЧНОЇ ІНФОРМАЦІЇ

2.1. ПОНЯТТЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОСОБЛИВОСТІ КАТЕГОРІЇ СОЦІАЛЬНО-ЕКОНОМІЧНА ІНФОРМАЦІЯ

Під інформацією розуміють відомості про явища, процеси та предмети навколишнього світу, і вона є одним з ресурсів, що використовуються людиною в трудовій діяльності та побуті. Але дані не завжди несуть у собі безпосередньо ту інформацію, яка необхідна для тих або тих цілей. Для того, щоб отримати необхідну інформацію, дані повинні підлягати певній обробці. Складність обробки даних залежить як від їх обсягу, так і від складності відповідних алгоритмів. У наш час обсяги інформації, що циркулює у виробничій, науковій, соціальній сферах, швидко зростають у зв'язку з різноманіттям зв'язків між об'єктами, яких ці дані стосуються. При цьому одні й ті самі дані можуть мати багатьох споживачів. І тоді на допомогу приходять комп'ютерні інформаційні системи, що можуть здійснювати обробку, зберігання та перетворення даних. Сукупність засобів, що забезпечують інформацією певне коло користувачів, називають інформаційною системою.

Користувачами інформаційних систем є люди, що використовують систему в своїй професійній діяльності і знайомі з основами комп'ютерної грамотності. Це можуть бути інженери, економісти, наукові працівники, різний управлінський персонал та ін. Більшість інформаційних систем зберігає дані, що описують предметну область у різних аспектах.

Предметна область — це сукупність інформаційних процесів і їх характеристик, що протікають у рамках об'єкта, вибраного для автоматизації, тобто це інформаційний бік функціонування автоматизованої системи, що відображає множину об'єктів і зв'язків між ними. Моделювання процесів предметної області передбачає її формальний опис у вигляді четвірки множин

$$A = \{x, y, z, d\}.$$

Множина x являє собою множину вхідних (початкових) даних, які виникають у цій предметній області $x = \{x_i\}$.

Множина y — це множина вихідної (підсумкової) інформації $y = \{y_i\}$.

Множина z — множина проміжної інформації $z = \{z_i\}$.

Множина d — сукупність аналітичних виразів, що встановлюють взаємозв'язок між множиною x, y, z, $d = \{d_i\}$.

На першому етапі моделювання предметної області відбувається опис існуючої системи керування об'єктом, при цьому основна увага приділяється технології обробки інформації.

На другому етапі проектується інформаційна база (x, y, z) нової інформаційної системи і проводиться докладний опис кожного елемента бази, структуризація елементів.

На третьому етапі формується модель предметної області, тобто визначається множина d, яка задає алгоритмічний опис інформаційних процесів.

Кожна сфера людської діяльності має свою інформацію, відображення суспільно-економічних явищ і процесів складає сутність економічної інформації на відміну, наприклад, від технічної чи біологічної. Економічна інформація [economic information] — інформація про суспільні процеси виробництва, розподілу, обміну та споживання матеріальних благ.

У рамках предметної області організації керування й економіки потрібно враховувати такі особливості економічної інформації:

1) дискретний характер інформації;

2) великі обсяги вхідних даних і результатної інформації;

3) складну структуру та ієрархічну взаємопідпорядкованість структурних елементів інформаційної бази;

4) значне переважання в процесі обробки логічних операцій над арифметичними (пошук, порівняння, об'єднання);

5) простоту і типовість арифметичних операцій;

6) циклічність обробки інформації, тобто застосування одних і тих самих аналітичних виразів для обробки інформації різнорідних об'єктів;

7) періодичність обробки інформації — виконання повного циклу обробки інформації через певні проміжки часу;

8) значні обсяги умовно-постійної інформації;

9) широке використання документарної форми представлення інформації;

10) наявність у процесі обробки інформаційних об'єктів двох типів: реквізитів-ознак, які описують якісні характеристики об'єктів, і реквізитів-основ, які являють собою кількісне представлення реквізитів-ознак;

11) обробка економічної інформації здійснюється, як правило, порціями (мінімальною порцією може бути рядок документа).

Властивості економічної інформації слід брати до уваги при розробці комп'ютерних інформаційних систем, при визначенні вимог до всіх видів забезпечення цих систем.

Обробка соціально-економічної інформації за допомогою ЕОМ вимагає її обов'язкової класифікації та кодування реквізитів ознак. Класифікація — система розподілу об'єктів (предметів, явищ, процесів, понять) за класами відповідно до певних ознак. Кожен об'єкт класифікації характеризується рядом ознак, які називаються ознаками класифікації. У процесі класифікації утворюються множини і підмножини за однією чи кількома ознаками, які називають класифікаційними угрупованнями. Класифікатор — це систематизований перелік найменувань класифікаційних угруповань з їх кодами, які являють собою умовні позначення, присвоєні за певними правилами.

Сукупність правил, за якими відбувається призначення кодів, називають системою кодування. Найчастіше в задачах обробки економічної інформації використовують систему кодування:

1) порядкову. Кодування реквізитів-ознак, при якому всі значення, що кодуються, зведені в список і кодовою комбінацією кожного значення є його порядковий номер у списку;

2) серійну або серійно-порядкову. Порядкове кодування, при якому послідовність порядкових номерів-кодів поділяється на групи-серії, що об'єднують об'єкти за якоюсь ознакою.

3) позиційну. Спосіб кодування реквізитів ознак, що набувають фіксоване число значень, при якому довжина кодової комбінації встановлюється рівною числу можливих значень реквізиту.

2.2. ТИПИ ПРОЦЕДУР ОБРОБКИ СОЦІАЛЬНО-ЕКОНОМІЧНОЇ ІНФОРМАЦІЇ

Розв'язання економічних задач вимагає обробки великих обсягів інформаційних сукупностей, що мають складну ієрархічну структуру, різноманітну форму представлення (числа, тексти, графіки, таблиці й т. ін.), різний тимчасовий цикл накопичення, зберігання й доступу до компонентів. Економічна інформація звичайно розташовується в економічних документах і являє собою сукупність різнотипних даних різної розмірності, що перебувають в ієрархічній залежності одні від одних. Логічний взаємозв'язок між окремими елементами економічної інфор-

мації встановлюється в кожному рядку документа, що підлягає обробці за одним і тим самим алгоритмом.

Дані для зберігання в пам'яті комп'ютера завжди певним чином організуються, тобто розчленовуються на окремі частини, що відносяться до певної предметної області. Сукупність згрупованих інформаційних відомостей, розміщену на зовнішньому носії інформації, звичайно називають набором даних. Набір даних, що зберігаються на зовнішньому носії, заведено називати файлом. Обмін даними між зовнішніми носіями й основною пам'яттю здійснюється не по одному знаку, а більшим одиницями — блоками. Блок як одиницю передачі даних називають фізичним записом. Блок може містити кілька логічних записів. Логічний запис файла являє собою набір логічно пов'язаних даних (полів) і є одиницею обміну між програмою та буфером введення-виведення.

Файл даних містить дані про множину однотипних об'єктів предметної області. Кожному об'єкту притаманний ряд характерних для нього властивостей (ознак, параметрів). Наприклад, властивості об'єкта СТУДЕНТ: ПРІЗВИЩЕ, І'МЯ, ПО БАТЬКОВІ, ФАКУЛЬТЕТ, СПЕЦІАЛЬНІСТЬ, КУРС, ГРУПА, ДАТА НАРОДЖЕННЯ, РІК ЗАКІНЧЕННЯ ШКОЛИ, СТАТЬ, НАЦІОНАЛЬНІСТЬ та ін. Сукупність усіх даних, що характеризують певний об'єкт (або процес, сутність і т. ін.) з даної множини, є записом файла.

Властивості об'єкта відображаються за допомогою змінних величин, які є елементарними одиницями інформації і називаються атрибутами або реквізитами. *Атрибут (attribute)* — це логічно неподільний елемент, що відноситься до властивості певного об'єкта чи процесу. Для кожного атрибута визначається множина значень. Кожний атрибут має певне значення, яке використовується в процесах обробки. Так, атрибут ДЕНЬ ТИЖНЯ може мати сім значень, а атрибут ОЦІНКА — чотири значення.

Під час проектування структури файла й обробки інформації атрибути поділяються на атрибути-ознаки та атрибути-основи. *Структура файла [file structure]* — це опис файла, що включає визначення структури записів файла, порядок їх розміщення й доступу до них. *Структура даних* — це множина елементів даних, об'єднаних і впорядкованих одним з прийнятих способів. *Атрибути-ознаки* (ключі) є якісною характеристикою об'єкта і звичайно беруть участь у логічних операціях, таких, як сортування, порівняння, компонування, редагування. Як атрибути-ознаки можуть виступати, наприклад, НОМЕР ЗАЛІКОВОЇ КНИЖКИ,

ПРІЗВИЩЕ, ФАКУЛЬТЕТ, КАФЕДРА, КУРС, ГРУПА тощо. *Атрибути-основи* характеризують кількісний бік об'єкта, залежать від атрибутів-ознак і беруть участь в обчислювальних операціях; це, наприклад, ВАГА, КІЛЬКІСТЬ, ВАРТІСТЬ, РІК. Атрибути-основи без атрибутів-ознак не дають визначення об'єкта. Кожен об'єкт характеризується набором атрибутів-ознак і атрибутів-основ. Будь-який документ навіть найскладнішої структури можна представити у вигляді атрибутів-ознак і атрибутів-основ.

У файлі звичайно виділяють один або декілька атрибутів, які відіграють роль ознаки, що дозволяє відрізняти і групувати записи, такий атрибут називають *ключем.* Ключ, що складається з одного атрибута, називається простим, ключ, що складається з кількох атрибутів, — складним. Так, атрибут ПРІЗВИЩЕ може бути простим ключем, а сукупність атрибутів ПРІЗВИЩЕ, ІМ'Я, ПО БАТЬКОВІ — складним.

Якщо для файла визначений ключ, то кожен запис характеризується значенням цього ключа. Для простого ключа це значення відповідного атрибута, а для складного ключа це впорядкована послідовність відповідних значень.

Інформаційна система повинна забезпечувати кожному користувачеві доступ лише до тієї частини даних, яка стосується його задач, і в той же час повинна захищати дані від несанкціонованого доступу, щоб уникнути читання секретних даних або їх створення. Тому в інформаційній системі може бути передбачена процедура захисту від несанкціонованого доступу. Найпростіший захист даних забезпечується за допомогою паролів. Паролем [password] називається код або секретне слово, що пред'являється користувачем системі для отримання доступу до даних і програм. Звичайно кожен клас даних може мати свій пароль, за яким дані можуть бути вибрані. Інший пароль може дозволити оновлення даних.

Будь-яка інформаційна система забезпечує користувачеві виконання таких функцій в узагальненому вигляді: введення даних в інформаційну систему; коригування даних; обробка даних; виведення даних з інформаційної системи.

Введення даних в інформаційну систему може здійснюватися в діалоговому чи пакетному режимі з магнітного диска, результатом введення завжди є файл на магнітному диску відповідної структури. У файл потрапляють лише відредаговані записи, що пройшли синтаксичний контроль.

Для того, щоб інформація, що вводиться до інформаційної системи, була коректною, її необхідно контролювати під час вве-

дення. Тому інформаційна система повинна включати засоби контролю інформації, що вводиться, які можна здійснювати програмно чи вручну, частіше за все — візуально.

Для виявлення помилок у вхідних даних можна використати різні методи синтаксичного контролю:

контроль діапазону;
контроль логічних обмежень;
контроль за довідником;
контроль типу даних.

Контроль діапазону означає, що деякі вхідні дані не можуть бути меншими або більшими від якихось заздалегідь відомих значень. Наприклад, під час введення стипендії відомі мінімальний і максимальний розмір стипендії за категоріями студентів.

Контроль логічних обмежень означає, що вхідні дані ніколи не приймуть деяких значень. Наприклад, під час введення таких даних, як місяць або день, можна ввести перевірку, що показує, що число місяців не може бути більшим 12, а днів — 28, 29, 30, 31.

Під час *контролю за довідником* використовуються довідники, що містять значення, які можуть приймати дані, що вводяться. Довідники оформляються у вигляді окремих файлів. Скажімо, під час введення коду факультету здійснюється перевірка з довідником факультетів, який містить коди і найменування всіх факультетів університету і послідовно переглядається під час введення факультету. Таких довідників в інформаційній системі може бути декілька.

Під час контролю типу даних необхідно контролювати, щоб значенню, оголошеному як символьне, відповідали символи, а оголошеному як цифрове — тільки цифри, і т. д.

Звичайно дані вводяться в файл з кількох полів, тому перевірку можна передбачити після введення кожного поля, цілого запису чи групи записів.

Під час введення великих обсягів інформації збільшується ймовірність появи помилок. Для своєчасного їх виявлення можна ввести спеціальні поля, що містять контрольні суми. Контрольна сума являє собою суму всіх числових атрибутів рядка документа, що підлягають контролю, розраховується за допомогою калькулятора й записується як окремий атрибут документа в кінці рядка, вводиться як і інші атрибути. Під час введення даних може бути здійснене повторне підсумовування програмою і порівняння отриманого результату з отриманою контрольною сумою. Рівність сум при цьому свідчить про правильність введення.

Якщо під час введення були виявлені помилки, слід видавати про це повідомлення в зручному вигляді для їх швидкого пошуку й виправлення. Потім введення даних повторюється для виправлення помилок.

Коригування даних може проводитися на рівні запису загалом або окремих його атрибутів, він включає операції зміни (оновлення), додавання та вилучення. Під час додавання і зміни записів або полів необхідно проводити синтаксичний контроль даних, що вводяться. Коригування в сучасних інформаційних системах, як правило, здійснюється в діалоговому режимі, при якому користувач системи в процесі діалогу сам визначає необхідні операції коригування.

Під час роботи практично з будь-якими документами пошук потрібних даних легше здійснювати тоді, коли документи впорядковані. Процедуру впорядкування називають ще сортуванням.

Під сортуванням даних прийнято розуміти процедуру розподілу записів за групами відповідно до певних правил або згідно з ознаками, що використовуються для ідентифікації запису — ключами. Можна здійснювати сортування безпосередньо під час введення даних, але цей процес здійснюватиметься повільно й це може призвести до проблем, якщо ознака, за якою здійснюється сортування, введена неправильно. Найкраще проводити сортування, коли дані введені, перевірені й записані на диск. Виконання сортування вимагає значних ресурсів часу і пам'яті. У залежності від характеру обробки інформації розрізняють такі способи: розподіл, упорядкування, підбір, вибірка та об'єднання.

Сортування способом розподілу передбачає поділ записів, що сортуються, на групи, які характеризуються однаковим значенням ознаки, що сортується. При такому способі записи з однаковим значенням ознаки, що сортується, збираються в одну групу, причому порядок розміщення записів усередині групи не має значення.

Під час сортування способом упорядкування записи розташовуються в порядку зростання або спадання значень ознаки, що сортується.

Сортування способом підбору означає знаходження для кожного запису з однієї групи даних відповідного запису з таким же значенням ознаки, що сортується, з іншої групи даних. При цьому записи обох груп даних можуть бути впорядковані або не впорядковані.

Сортування способом вибірки призначене для відбору з групи даних записів, що характеризуються певними значеннями ознаками, наприклад, вибір записів, що належать одному і тому самому об'єкту.

Сортування методом об'єднання передбачає отримання однієї послідовності записів, побудованої за певним правилом, з двох і більше послідовностей, побудованих за тим же правилом.

Процедура обробки даних. У загальному випадку розв'язання задачі обробки даних вимагає перегляду одного або кількох файлів, виконання деяких операцій над даними записів і формування вихідного документа чи вихідного файла.

Для того, щоб отримати вихідні файли з вхідних, необхідно виконати певну послідовність дій, яку називають процедурою обробки. В обробці даних є процедури, типові для всіх задач. Однією з них є вибірка. Ця процедура використовується для виділення з початкових файлів потрібної інформації. Вибірка даних відбувається швидше, якщо записи початкового файла впорядковані за тим ключем, за яким відбувається вибірка.

Нехай, наприклад, є екзаменаційна відомість, дані якої знаходяться в файлі EKZ (табл. 2.1), потрібно отримати відомість, що в ній вказані всі студенти, які склали іспит на «відмінно».

Таблиця 2.1

ФАЙЛ EKZ

Прізвище	Дисципліна	Оцінка
Андрієнко	Вища математика	5
Андрієнко	Комп'ютерна техніка	4
Андрієнко	Статистика	5
Андрієнко	Соціологія	4
Безбородько	Вища математика	5
Безбородько	Комп'ютерна техніка	5
Безбородько	Статистика	4
Безбородько	Соціологія	4
Василенко	Вища математика	5
Василенко	Комп'ютерна техніка	5
Василенко	Статистика	5
Василенко	Соціологія	5

Часто-густо в повсякденному житті виникає необхідність вибирати лише один запис з даним значенням ключа. Наприклад, нас може цікавити, яку оцінку має студент Василенко з вищої математики.

Вибірка є основною процедурою для будь-якої інформаційної системи. В таких системах людина для отримання необхідної інформації має сформулювати запит, у якому задаються умови вибірки, тобто вказуються ті значення величин, за якими проводитиметься відбір інформації, або вказується діапазон таких величин. Умови вибірки можуть бути простими або складними. Прості умови задаються за допомогою операцій відносин: більше ніж, менше ніж, рівно, не рівно, більше або рівно, менше або рівно. Скажімо, умова «оцінка = 5» є простою умовою. Складні умови являють собою комбінацію простих умов, сполучених логічними зв'язками «І», «АБО», «НЕ». Наприклад, умова «оцінка = 5 АБО оцінка = 4» є складною умовою, що передбачає вибірку студентів, які склали іспит без «трійок».

Процедура вибірки є процедурою для отримання вихідних даних з одного вхідного. Проте в багатьох задачах обробки даних виникає необхідність отримання вихідних даних з декількох вхідних файлів. Якщо вхідні файли складаються з одних і тих самих величин і однаково впорядковані за одним і тим самим ключем, то для отримання результуючого файла, що включає записи початкових масивів, використовується процедура злиття. Злиття — об'єднання двох або більше файлів в один файл. Після злиття результуючий файл буде також упорядкований, як упорядковані вхідні. Процедура злиття часто-густо використовується при накопиченні однотипної інформації, що надходить з різних джерел у послідовні проміжки часу.

Якщо початковий файл нарівні із загальними має різні величини, то для отримання результуючого файла, що містить дані з кількох файлів, використовується процедура з'єднання файлів. Під час з'єднання двох файлів за ключем множина значень ключових величин результуючого файла дорівнює перерізу множин значень ключових величин вхідного файла. Процедуру з'єднання використовують у тих випадках, коли необхідно зібрати в один файл різні дані про одні й ті самі об'єкти, що знаходяться в різних вихідних файлах.

Розглянуті вище процедури переносили потрібні дані без змін до результуючого файла, тобто результуючий файл містить лише ті дані, які є в вхідному файлі. Для більшості задач обробки даних вихідні дані повинні містити величини, значення яких відсутні в початкових даних, але які можна отримати із значень величин початкових даних, проробивши з ними певні обчислення. Дані для обчислень можуть братися з одного запису, з множини значень однієї і тієї самої величини з усіх записів або ж частини

записів. У першому випадку використовується процедура обчислення всередині запису, в другому — обчислення всередині файла. Під час виконання процедури обчислення всередині запису може бути обчислена не одна нова величина, а декілька, які виконуються за певними формулами.

Кожен початковий файл можна умовно розбити на групи записів, що мають одну й ту саму властивість. Наприклад, записи, що відносяться до однієї і тієї ж групи, до одного курсу, до одного факультету. В свою чергу, групи записів можуть бути розбиті на підгрупи, що визначаються якоюсь ознакою.

Під час виконання процедури обчислення всередині файла аргументами для отримання підсумкових значень служать значення величин, що входять до різних записів певної групи (підгрупи) записів початкового файла. Дана процедура використовується для отримання підсумкових значень певних величин у групах записів, що мають один і той самий ключ, або по всьому файлу загалом. Звичайно до таких підсумкових значень відносяться: кількість, сума, середнє і т. ін.

Зазвичай під час виконання процедури крім підсумкових значень у групах записів початкового файла обчислюється також підсумкове значення по всьому файлу загалом, цю процедуру називають агрегуванням даних.

Незважаючи на відмінність варіантів розв'язання тієї або тієї задачі в процедурі обробки економічної інформації можна виділити дві функції: обробку запиту й отримання звіту чи відомості. Обробка запиту передбачає відбір записів з файла відповідно до заданої умови. Наприклад, виведення студентів, які народилися в поточному місяці, з зазначенням дати народження. Або виведення відмінників другого курсу факультету інформаційних систем і технологій. Формування звіту (відомості) відрізняється від запиту тим, що звіт охоплює не частину файла, як запит, а його цілком; при формуванні звіту інформація, як правило, обробляється. Тому звіт не тільки відображає зміст файла, а й певним чином аналізує його.

Процес обробки інформації завершується видачею вихідних документів на дисплей або на принтер, а також записом у файл вихідних даних. На дисплей частіше виводяться дані в формі, близькій до форми їх зберігання. Обсяги даних, що виводяться при цьому, як правило, невеликі. Значна частина результатів обробки даних формується у вигляді друкованих документів.

Приклади алгоритмів типових процедур обробки соціально-економічної інформації наведені в додатку.

Розділ 3

ЕТАПИ РОЗВ'ЯЗУВАННЯ ЗАДАЧІ З ВИКОРИСТАННЯМ ПЕОМ

3.1. ЖИТТЄВИЙ ЦИКЛ ПРОГРАМНОГО ВИРОБУ

Поняття розв'язання задачі з допомогою ПЕОМ не обмежується тільки етапом обчислень, що проводяться на ПЕОМ. Тому необхідно виконати певну підготовчу роботу, різноманітну за характером. Цю роботу можна розбити на етапи, причому перехід до кожного наступного етапу можливий лише після того, як будуть виконані всі попередні.

Процес розробки та використання програмного виробу називається його життєвим циклом. *Життєвий цикл* — це безперервний процес, що починається з моменту прийняття рішення про необхідність створення програмного виробу і закінчується в момент його повного вилучення з експлуатації. Основні етапи циклу: проектування, програмування, налагодження, дослідна експлуатація, зберігання, промислова експлуатація та супроводження.

Структура життєвого циклу за стандартом ICO/IEC 12207 базується на трьох групах процесів:

- основні процеси життєвого циклу (придбання, постачання, розробка, експлуатація, супроводження);
- допоміжні процеси, що забезпечують виконання основних процесів (документування, керування конфігурацією, забезпечення якості, верифікація, атестація, оцінка, аудит, розв'язання проблем);
- організаційні процеси (управління проектами, створення інфраструктури проекту, визначення, оцінка й поліпшення самого життєвого циклу, навчання).

Розробка включає в себе всі роботи зі створення програмного забезпечення та його компонентів відповідно до заданих вимог, навіть оформлення проектної та експлуатаційної документації, підготовку матеріалів, необхідних для перевірки працездатності і відповідної якості програмних продуктів, матеріалів, необхідних для організації навчання персоналу, і т. ін. Розробка програмного забезпечення включає в себе аналіз, проектування, кодування (програмування), тестування.

Експлуатація включає в себе роботи з впровадження компонентів програмного забезпечення в експлуатацію, в тому числі

конфігурування бази даних і робочих місць користувачів, забезпечення експлуатаційною документацією, проведення навчання персоналу і т. ін., і безпосередньо експлуатацію, в тому числі локалізацію проблем і усунення причин їх виникнення, модифікації програмного забезпечення в рамках встановленого регламенту, підготовку пропозицій до вдосконалення, розвитку й модернізації системи.

Управління проектом пов'язане з питаннями планування й організації робіт, створення колективів розробників і контролю за термінами і якістю виконання завдань. Технічне та організаційне забезпечення проекту включає вибір методів і інструментальних засобів для реалізації проекту, визначення методів опису проміжних станів розробки, розробку методів і засобів випробувань програмного забезпечення, навчання персоналу й т. ін. Забезпечення якості проекту пов'язане з проблемами верифікації, перевірки й тестування програмного забезпечення. Верифікація — це процес визначення того, чи відповідає поточний стан розробки, досягнутий на даному етапі, вимогам цього етапу. Перевірка дає можливість оцінити відповідність параметрів розробки початковим вимогам. Перевірка частково збігається з тестуванням, пов'язаним з ідентифікацією відмінностей між дійсними та очікуваними результатами й оцінкою відповідності характеристик програмного забезпечення початковим вимогам. У процесі реалізації проекту важлива роль належить питанням ідентифікації, опису та контролю конфігурації окремих компонентів і всієї системи загалом.

Керування конфігурацією є одним з допоміжних процесів, що підтримують основні процеси життєвого циклу програмного забезпечення, передусім процеси розробки й супроводження програмного забезпечення. При створенні проектів складних інформаційних систем, що складаються з багатьох компонентів, кожний з яких може мати різновиди або версії, виникає проблема обліку їхніх зв'язків і функцій, створення уніфікованої структури та забезпечення розвитку цілої системи. Керування конфігурацією дозволяє організувати, систематично враховувати і контролювати внесення змін до програмного забезпечення на всіх стадіях життєвого циклу.

Кожен процес характеризується певними задачами і методами їх вирішення, початковими даними, отриманими на попередньому етапі, і результатами. Життєвий цикл програмного забезпечення носить ітераційний характер: результати чергового етапу часто викликають зміни в проектних рішеннях, здійснених на більш ранніх етапах.

Життєвий цикл починається з розуміння необхідності та потреби в такому програмному виробі, з вивчення та визначення завдання. Після того, як сформулювана постановка задачі, починається проектування. Воно означає визначення структури даних, алгоритмів розв'язання задачі, структури програми та порядку взаємодії складових компонентів і комплексів програмного виробу. Кожен компонент кодується мовою програмування, налагоджується спочатку окремо, а потім у комплексі, після цього програмний виріб є готовим до експлуатації. У процесі експлуатації можуть бути виявлені помилки, які не були знайдені раніше. У зв'язку зі зміною умов застосування користувач усвідомлює потребу або в модернізації програми, або розробки цілковито нової програми — і цикл повторюється.

Життєвий цикл поділяється на стадії. Першою стадією є *технічне завдання*. На цій стадії мають бути визначені вимоги до програми, яка розробляється. Для цього виконують аналіз аналогічних систем, що існують, проводять дослідження здійснювання та обґрунтовують ефективність програми, яка розробляється.

У цей же час необхідно з'ясувати істотні обмеження на програму (вони повинні бути сформульовані докладно) та встановити їх пріоритетність. Також необхідно виробити компроміс між конфліктуючими вимогами.

Наступною стадією є *пояснювальна записка*. Ця стадія робіт присвячена зовнішньому проектуванню програмного виробу, розробці його архітектури. Зовнішні специфікації, які є результатом зовнішнього проектування, описують очікувану поведінку програмного виробу з погляду користувача, тобто всі можливі вхідні дані системи — як допустимі, так і ні, та відповідну реакцію системи на ці дані. На цьому етапі починається розробка тестів усього виробу.

Розробка архітектури програмного виробу є процесом розбивки великої системи на менші частини: програми, підпрограми, модулі, функції.

Технічний проект — третя стадія життєвого циклу програми. На цій стадії робіт здійснюється проектування архітектури програми, яка включає визначення всіх модулів програми, їх взаємозв'язки.

Модуль — це послідовність логічно пов'язаних фрагментів, що оформляються як окрема частина програми. Модуль має відповідати таким вимогам:

кожен модуль може компілюватися окремо;

модулі незалежні один від одного, тобто змінювання в одному модулі не вимагає змін в інших;

модуль реалізує одну функцію або групу пов'язаних функцій.

Робоче проектування. Дана стадія завершує розробку програми. Виконується кодування алгоритму мовою програмування, здійснюється індивідуальне налагодження модулів, об'єднання їх у комплекс і налагодження всього комплексу, перевірка роботоздатності за допомогою розроблених тестів.

Налагодження [debugging] — процес виявлення, локалізації та усунення помилок у програмі.

Суть налагодження полягає в тому, що користувач підготовляє систему тестів, з допомогою якої перевіряється робота програми в різних можливих режимах. Кожен тест містить набір початкових даних, результат яких відомий. Якщо в результаті роботи програми з даним тестом виходять результати, які відрізняються від очікуваних, то це свідчить про наявність помилки. Тест необхідно побудувати так, щоб він не лише встановлював сам факт помилки, а й локалізував цю помилку, тобто звузив частину, що підозрюється, програми, яка містить помилки.

Налагодження великих і складних програм є процесом трудомістким внаслідок складності їх структури, тому слід підбирати не один, а кілька тестів, щоб перевірити найбільше число можливих ситуацій, які можуть трапитися в процесі роботи програми. Тож кожна програма під час налагодження вимагає ретельного підбору системи тестів.

Налагодження починається із запуску програми з найпростішими даними, що вводяться. Потім перевіряється, як програма поводиться при максимальних і мінімальних значеннях, що вводяться, потім може бути здійснена перевірка поведінки програми при введенні значень, на які вона не розрахована. Потім здійснюється перевірка даних, що виводяться програмою.

На стадії *впровадження* виконуються підготовка та передача програми та програмної документації для супроводження користувачеві.

Стадія експлуатації не є стадією розробки, виконуються роботи, які пов'язані з виправленням програмних дефектів або з незначними змінами в програмі.

Держстандарт 19.102—77 «ЕСПД. Стадії розробки» передбачає такі стадії розробки: технічне завдання, ескізний проект, технічний проект, робоче проектування, впровадження.

На стадії технічного завдання виділяють такі етапи:

1) обґрунтування необхідності розробки програми: постановка задачі; збір вихідних даних; вибір і обґрунтування критеріїв ефективності та якості програми, що розробляється; обґрунтування необхідності проведення науково-дослідних робіт;

2) науково-дослідні роботи: визначення структури вхідних і вихідних даних; попередній вибір методів розв'язання задачі; обґрунтування доцільності застосування програм, які розроблені раніше; визначення вимог до технічних засобів; обґрунтування принципової можливості виконання поставленого завдання;

3) розробка та затвердження технічного завдання: визначення вимог до програми; розробка техніко-економічного обґрунтування розробки програми; визначення стадій, етапів і термінів розробки програми та документації до неї; вибір мови програмування; визначення необхідності проведення науково-дослідних робіт на наступних стадіях; погодження та затвердження технічного завдання.

Стадія ескізного проекту складається з таких етапів:

1) розробка ескізного проекту: попередня розробка структури вхідних і вихідних даних; уточнення методів розв'язання задачі; розробка загального опису алгоритму розв'язання задачі; розробка техніко-економічного обґрунтування;

2) затвердження ескізного проекту; розробка пояснювальної записки; погодження та затвердження ескізного проекту.

На стадії технічного проекту слід виділити такі етапи:

1) розробка технічного проекту: уточнення структури вхідних і вихідних даних; розробка алгоритму розв'язання задачі; визначення форм подання вхідних і вихідних даних; визначення семантики та синтаксису мови; розробка структури програми; остаточне визначення конфігурації технічних засобів;

2) затвердження технічного проекту: розробка плану заходів з розробки та впровадження програм; розробка пояснювальної записки; погодження та затвердження технічного проекту.

На стадії робочого проекту визначаються такі етапи:

1) розробка програми: розробка та налагодження програми;

2) розробка програмної документації: розробка програмної документації відповідно до вимог Держстандарту 19.101—77;

3) випробування програми: розробка, погодження та затвердження програми та методики випробувань; проведення попередніх державних, міжвідомчих, приймально-здавальних та інших видів випробувань; коригування програми та програмної документації за результатами випробувань.

На стадії впровадження здійснюється підготовка та передача програми: підготовка та передача програми та програмної документації для супроводження та (або) виготовлення; оформлення та затвердження акта про передачу програми на супроводження та (або) виготовлення; передача програми у фонд алгоритмів і програм.

Для використання програмних засобів, крім власне програми, необхідно створювати повну та якісну програмну документацію. Програмні документи містять відомості, необхідні для розробки, виготовлення, експлуатації та супроводження програм. До них відносять специфікацію, відомість тримачів оригіналів, текст програми, опис програми, програму та методику випробувань, технічне завдання, пояснювальну записку, експлуатаційні документи.

Експлуатаційні документи містять відомості для забезпечення функціонування та експлуатації програми. До них відносять: відомість експлуатаційних документів, формуляр, опис застосування, посібник системного програміста, посібник програміста, посібник користувача, опис мови, посібник з технічного обслуговування.

3.2. СУЧАСНА МЕТОДОЛОГІЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Стандарт ICO/IEC 12207 не пропонує конкретну модель життєвого циклу і методи розробки програмного забезпечення, він описує структуру процесів життєвого циклу програмного забезпечення (ЖЦ ПЗ), але не конкретизує в деталях, як реалізувати або виконати дії та задачі, включені в ці процеси. Під моделлю ЖЦ розуміємо структуру, що визначає послідовність виконання і взаємозв'язки процесів, дій і завдань, що виконуються протягом життєвого циклу. Модель життєвого циклу залежить від специфіки інформаційної системи і специфіки умов, у яких остання створюється і функціонує. Виділяють такі моделі життєвого циклу:

традиційна (кустарна);

модель швидкого прототипу;

модель, що базується на повторному використанні компонентів;

модель, що базується на автоматизованому синтезі програм;

каскадна;

спіральна;

CASE-метод.

Згідно з традиційною моделлю кодування та налагодження програми виконуються на початку розробки, а потім програму доводять до вимог користувача. В традиційній моделі відсутні етапи аналізу та проектування, і, як наслідок, програмний виріб має низькі характеристики якості, висока трудомісткість модифікації таких програм, процесу управління проектом взагалі немає.

Метод швидкого прототипу передбачає розробку в стислі строки діючого макета найкритичнішої до вимог користувача частини програмного забезпечення і проведення дослідної експлуатації макета перед тим як перейти до розробки повномасштабного зразка. Зазвичай насамперед прототипуванню підлягає інтерфейс користувача з майбутньою системою. Це дозволяє залучити кінцевих користувачів до активної співпраці на початкових стадіях розробки програми і таким чином уникнути доробок уже закінченої системи, що потребує значних коштів. Основне призначення даної моделі — полегшити виявлення всіх вимог користувача. Тому, як правило, прототип після розробки технічного завдання більше не використовується і решта моделі життєвого циклу збігається з каскадною.

Модель, що базується на повторному використанні компонентів, є основою так званого монтажного програмування, яке дозволяє суттєво скоротити тривалість розробки програмного забезпечення, підвищити надійність при одночасному скороченні затрат на супроводження. Найбільшого ефекту під час використання цієї моделі досягається в тих випадках, коли значну частину завдань можна сформулювати в термінах порівняно невеликої кількості підзавдань, які відповідають стандартним підпрограмам. Тоді розробка програмного забезпечення зводиться до написання порівняно нескладної програми, яка викликає ці підпрограми в потрібній послідовності та організує обмін даними між ними. Такий підхід передбачає вивчення понять і відношень відповідної області і розробку пакета прикладних програм, що реалізує ці поняття і відношення, а також вивчення технології застосування цього пакета під час формалізації завдання. Однак унікальні алгоритми обробки інформації складних завдань за допомогою стандартних програм описати практично неможливо. Тому модель, що базується на повторному використанні компонентів, у чистому вигляді не застосовується.

Модель, що базується на автоматизованому синтезі програм, базується на представленні знань як про предметну область, так і про процеси створення програмних засобів. Реалізація цієї моделі потребує достатньо високих початкових витрат на побудову моделей знань та створення інструментальних засобів для їх підтримки, що пов'язано з ризиком значного подорожчання розробки.

Основною характеристикою каскадної моделі є розбиття всієї розробки на етапи, причому перехід від одного етапу до наступного відбувається тільки після того, як буде повністю завершена робота на поточному (рис. 3.1.). Кожен етап завершується випус-

ком повного комплексу документації, достатньої для того, щоб розробка могла бути продовжена іншою командою розробників.

Каскадний підхід найбільш придатний для побудови інформаційних систем, для яких на самому початку розробки можна досить точно і повно сформулювати всі вимоги, щоб надати розробникам можливість реалізувати їх якнайкраще. До цієї категорії належать складні розрахункові системи, системи розв'язання задачі в режимі реального часу.

Однак, у процесі використання цього підходу виявилося, що реальний процес створення програмного забезпечення ніколи повністю не вкладався в таку жорстку схему, постійно виникає потреба в поверненні до попередніх етапів і уточненні чи перегляді раніше прийнятих рішень.

Основним недоліком каскадного підходу є істотне запізнювання з отриманням результатів. Узгодження результатів з користувачами виконується тільки після завершення кожного етапу робіт, вимоги до інформаційної системи у вигляді технічного завдання на весь час її створення не змінюються. Таким чином, користувачі можуть внести свої зауваження лише після того, як робота над системою буде повністю завершена. У разі неточного викладу вимог або зміни протягом тривалого періоду створення інформаційної системи, користувачі отримують систему, що не задовольняє їхніх потреб.

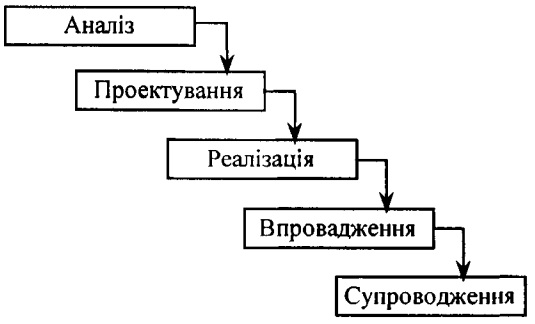


Рис. 3.1. Каскадна модель розробки програмного забезпечення

Для подолання недоліків каскадної моделі була запропонована спіральна модель життєвого циклу, що робить наголос на початкових етапах життєвого циклу: аналіз і проектування. На цих етапах можливість реалізації проектних рішень перевіряється шляхом створення прототипів. Кожен виток спіралі відповідає створенню фрагмента чи версії програмного забезпечення, на

ньому уточнюються цілі та характеристики проекту, визначається його якість і плануються роботи наступного витка спіралі (рис. 3.2.). Таким чином заглиблюються і послідовно конкретизуються деталі проекту, й у результаті вибирається обґрунтований варіант, який доводиться до реалізації.

Рис. 3.2. Спіральна модель життєвого циклу

Розробка ітераціями відображає об'єктивно існуючий спіральний цикл створення системи. Неповне завершення робіт на кожному етапі дозволяє перейти до наступного етапу, не чекаючи повного завершення робіт на поточному. При ітераційному способі розробки роботу, якої бракує, можна буде виконати на наступній ітерації. А головним завданням є якнайшвидше показати користувачам системи працездатний продукт, тим самим активізуючи процес уточнення та доповнення вимог.

Основним недоліком спіральної моделі є визначення моменту переходу на наступний етап. Для цього визначення необхідно ввести тимчасові обмеження на кожний з етапів життєвого циклу. Перехід здійснюється відповідно до плану, навіть якщо не вся запланована робота закінчена. План складається на основі статистичних даних, отриманих у попередніх проектах, і особистого досвіду розробників. Ще одним недоліком спіральної моделі є значні витрати ресурсів під час розробки великих проектів. Тому цю модель використовують тоді, коли масштаби проекту невеликі, але існує значна невизначеність щодо вимог

користувачів. Якщо ж проект достатньо великий, то в ньому можна виділити обмежену за обсягом підсистему, яку, справді, доцільно розробляти, використовуючи спіральну модель. У зв'язку з труднощами планування робіт цю модель найчастіше застосовують у випадках, коли розробник і користувач — одна й та сама організація.

У теперішній час розробці програмного забезпечення притаманний високий ступінь автоматизації всіх робіт і перехід до індустріалізації цього процесу. Це виражається передусім у впровадженні CASE-систем, яке реалізує певну методологію розробки програмного забезпечення, що охоплює весь життєвий цикл програмного забезпечення. Термін CASE (Computer Aided Software Engineering) набув у наш час вельми широкого змісту. Початкове значення терміна CASE, обмежене лише питаннями автоматизації розробки програмного забезпечення, нині набуло нового змісту, який охоплює процес розробки складних інформаційних систем у цілому. Тепер під терміном CASE-засобу розуміють програмні засоби, що підтримують процеси створення й супроводження інформаційних систем, включаючи аналіз та формулювання вимог, проектування прикладного програмного забезпечення і бази даних, генерацію коду, тестування, документування, забезпечення якості, конфігураційне керування і керування проектом, а також інші процеси. Під засобами проектування інформаційних систем будемо розуміти комплекс інструментальних засобів, що забезпечують у рамках вибраної методології проектування підтримку повного життєвого циклу інформаційних систем. CASE-засоби разом із системним програмним забезпеченням і технічними засобами утворюють повне середовище розробки інформаційних систем. CASE-технологія являє собою методологію проектування програмного забезпечення, а також набір інструментальних засобів, що дозволяють у наочній формі моделювати предметну область, аналізувати цю модель на всіх етапах розробки й супроводження програмного забезпечення та розробляти програми відповідно до потреб користувачів. Більшість існуючих CASE-засобів базується на методологіях структурного або об'єктно-орієнтованого аналізу і проектування, що використовують специфікації у вигляді діаграм або текстів для опису зовнішніх вимог, зв'язків між моделями системи, динаміки поводження системи й архітектури програмних засобів. CASE-методологія припускає, як правило, наявність таких етапів: стратегічне планування, аналіз, проектування, реалізацію, впровадження, експлуатацію та супроводження. Кожен етап характеризується

певними задачами і методами їх розв'язання, початковими даними, отриманими на попередньому етапі, й результатами.

На етапі стратегічного планування розробляється узагальнена модель організації, що описує її інформаційні потреби і (або) процеси, що тривають у ній. Розроблена модель стає основою для процесу автоматизації даної організації та джерелом планування цього процесу. Планують розподіл усіх видів ресурсів, визначають послідовність робіт і їх пріоритети. Даний етап у CASE-системах відсутній. Результатом роботи на цьому етапі є концептуальна модель, що буде основою для етапу аналізу, і план розробки.

Етап аналізу починається з деталізування концептуальної моделі, на цьому етапі визначається, що, власне, підлягає розробці, і передбачається розробка сукупності моделей, які відображають різні аспекти проблеми. Процеси й потоки інформації відображаються діаграмами потоків даних, інформаційні потреби системи, що розробляється, — діаграмами суть-зв'язок, тимчасові аспекти функціонування — діаграмами станів-переходів. Усі три типи діаграм застосовуються і на етапі стратегічного планування, але на узагальненішому рівні.

На діаграмах потоків даних відображаються потоки даних, процеси, що перетворюють вхідні потоки у вихідні, сховища інформації, зовнішні по відношенню до системи. Кожен з процесів може бути представлений діаграмою більш низького рівня. Надалі ці діаграми служать основою для формування структури програмного забезпечення, що розробляється. На діаграмах суть-зв'язок показуються так звані сутності (інформація, що являє інтерес з погляду подальшого зберігання) і зв'язки між сутностями з відображенням характеру зв'язків. Ця діаграма є основою для проектування баз даних. На діаграмі станів-переходів відображаються стани, в яких може знаходитися система, і можливі переходи з одного стану в інший. На основі діаграми потім проектується інтерфейс користувача. Даний етап закінчується вибиранням попередньої архітектури системи з сукупності моделей, і ця інформація передається на наступний етап. Формується так званий словник даних — перелік описів усіх елементів моделей. У результаті отримуємо модель програмного забезпечення, що являє собою сукупність моделей потоків даних, суть-зв'язок, станів-переходів; словник даних з описом усіх елементів моделей: потоків даних, сховищ даних, процесів, сутностей і їхніх атрибутів, а також зв'язків між сутностями, станів і переходів; план тестування; попередню архітектуру системи.

Отримана інформація називається структурною специфікацією на програмне забезпечення, що розробляється, і відображає вимоги на його розробку.

Етап загального проектування починається з аналізу попередньої архітектури, сформованого на попередньому етапі. Аналіз ґрунтується на критеріях якості, визначених для архітектури. Архітектура має логічний характер у тому плані, що вона не прив'язана до конкретних мов програмування, операційних систем та особливостей комп'ютера. У результаті отримуємо: архітектуру системи у вигляді так званої структурної схеми, на якій відображені програмні модулі та зв'язки між ними, а також дані, що передаються від одного модуля до іншого; проект баз даних і інтерфейсу користувача; словник даних, що містить опис цих даних і їхньої структури; специфікації модулів з описом їх призначення та особливостей; проект архітектурних тестів.

Вся ця інформація передається на етап детального проектування. Тут розробляються алгоритми програмних модулів на основі їх специфікацій, отриманих на попередньому етапі, а також проекти модульних тестів з урахуванням їхньої структури, все це переходить на етап реалізації.

На етапі реалізації програмні модулі кодуються вибраною мовою програмування відповідно до розроблених алгоритмів і налагоджуються. Далі за планом тестування проводиться тестування програмної системи. Після цього розроблене програмне забезпечення передається на супроводження. Життєвий цикл розробки на цьому закінчується і починається життєвий цикл програмного забезпечення.

CASE-технологія — потужний засіб, при правильному використанні якого процес розробки інформаційних систем спрощується завдяки використанню певного комплексу програм (CASE-засобу) та нового підходу до методології та правил, за якими відбувається створення інформаційних систем розробниками. За CASE-технологією інформаційні системи створюються ітераційним шляхом, при якому кілька разів відбувається процес проходження одних і тих самих етапів розробки. Це стає можливим завдяки тому, що використовуються автоматизовані засоби збереження інформації про попередній стан розробки кожної з робіт, з яких складається процес розробки системи. При зміні у форматі або переліку вхідних або вихідних даних однієї зі складових для всіх інших, які пов'язані зі зміною, автоматично вставляється прапор, який вказує на те, що відбулася така зміна і розробники повинні звернути увагу на цей факт.

Але слід дуже обережно підходити до проблеми переходу на нову систему проектування проектів. Це пов'язано не тільки з проблемою вивчення можливостей та функцій нових програмних пакетів, за допомогою яких і відбувається процес розробки проекту, а й з тим, що проектувальниками витрачатимуться досить значні зусилля на вивчення нових технологій та методик проектування.

CASE-засоби забезпечують такі основні можливості з проектування та модифікації інформаційних систем:

1) швидкої побудови структурно-логічного опису складних комплексів та систем об'єктів, що взаємодіють між собою, динамічних процесів, потоків та баз даних під час побудови архітектури інформаційної системи, що моделюються. Швидкість розробки є принципово важливою, оскільки моделювання має бути не гальмом, а навпаки — прискорювачем процесу проектування системи;

2) представлення логічно-динамічних процесів обробки інформаційних і матеріальних потоків системи на різних рівнях деталізації архітектури з використанням узагальнень, які адекватні даним рівня деталізації. Можливо описувати одні й ті самі процеси з різною мірою деталізації, а також вносити до моделі алгоритми функціонування організації у вигляді бізнес-правил, що формулюються експертами. Під час опису архітектури інформаційної системи є можливість наводити в моделі узагальнені правила логічної організації процесів обробки інформації, обмеження та властивості архітектур, що моделюються;

3) прототипування динамічних моделей архітектур, тобто представлення їх у формі, яка є відкритою, модифікованою, деталізованою. Для великих організацій процес створення прикладних моделей може розпочинатися та паралельно розвиватися в різних точках. Тому необхідно забезпечити єдину форму створення моделей на базі одного прототипу. Така форма представлення моделей відіграє також роль прототипу архітектури інформаційної системи, після чого послідовно уточнюється та деталізується до отримання результативних рішень. Важливо, щоб цей процес не супроводжувався кожного разу перепрограмуванням моделей, а полягав у коригуванні та деталізації простих структурно-візуальних компонент, логічних зв'язків та правил функціонування моделей;

4) створення бібліотек типових об'єктів та моделей типових процесів, за допомогою яких можуть бути створені моделі нових проектів. Повинна бути забезпечена можливість інтеграції ре-

зультатів моделювання однієї з інформаційних систем, що взаємодіють між собою, до імітаційних експериментів з іншою інформаційною системою для спрощення процесу дослідження їх сумісного функціонування;

5) можливість інтелектуальної обробки результатів динамічного моделювання для оперативного отримання необхідних оцінок метрик та характеристик архітектури інформаційних систем, що досліджуються, за результатами імітаційних експериментів протягом великих відрізків модельного часу;

6) побудову динамічних моделей, які є орієнтованими на розробників інформаційних систем та експертів організацій, які виконують не лише формування та оцінку вимог до інформаційних систем, а й реінжиніринг бізнес-процесів організації, що автоматизується.

3.3. ТЕХНОЛОГІЯ ПРОГРАМУВАННЯ

У міру розвитку обчислювальної техніки виникали різні технології програмування. На кожному етапі створювався новий підхід, який допомагав програмістам давати раду дедалі більшому ускладненню програм. *Технологія програмування* [programming technology] — система методів, способів і прийомів розробки й налагодження програм.

Висхідне програмування (програмування «знизу вгору») [bottom up programming] — спосіб розробки програм, під час якого спочатку проектуються і налагоджуються програми для виконання простих операцій, а потім розроблені модулі об'єднуються в єдину програму. При цьому структура і функціональне призначення функцій більш високих рівнів витікає з функцій нижнього рівня.

Низхідне програмування (програмування «згори вниз») [top-down programming] — спосіб розробки програм, за якого на кожному кроці деталізування для кожної задачі складається програма в термінах виділених в ній підзадач. Алгоритм розв'язання задачі розбивається на простіші частини або підзадачі. Підзадачі виділяють так, щоб вони були незалежними. При цьому складають план розв'язання цілої задачі, пунктами якого і є виділені частини. План записують графічно, визначають головну і підлеглі підзадачі та зв'язки між ними, встановлюють, які дані отримує кожна підзадача для функціонування і які результати видає. Складається програма, яка містить виклики підпрограм (процедур або функцій), відповідних виділеним підзадачам. Цю програму можна відразу налагоджувати, підпрограми для підзадач тимчасово замінюються «заглушками». Аналогічно проводиться деталізація і програмування кожної підзадачі. Процес послідовної деталізації відбувається доти, доки не буде написана програма для кожного фрагмента алгоритму. При цьому на кожному етапі вироблення програми є діючий варіант програми, налагодження якої ведеться по ходу всієї її розробки.

Структурне програмування [structured programming] — конструювання програм, що використовує лише ієрархічно вкладені конструкції, кожна з яких має єдину точку входу та єдину точку виходу. Структурне програмування передбачає створення зрозумілих, локально простих і зручних до читання програм, характерними особливостями яких є модульність, використання уніфікованих структур слідування, вибору і повторення, відмова від неструктурованих передач керування, обмеження використання глобальних змінних.

Принципами структурного програмування є:

1. Низхідне програмування, під час якого задача розбивається на кілька підзадач, що програмуються у вигляді окремих модулів.

2. Використання під час програмування трьох структур керування ходом програми: слідування, вибір і повторення. З цих структур може бути побудована програма будь-якої підзадачі.

3. Відмова від безумовних передач керування. Будь-який програмний блок має єдиний вхід і єдиний вихід. Він може бути просто частиною програми, але може мати вигляд модуля, підпрограми або функцій. Блок завжди повертає керування оператору програми, наступному після того, який звернувся до нього.

Структурне програмування полегшує і прискорює написання великої програми, спрощує налагодження програми, допомагає поділу роботи між виконавцями і відкриває можливості для подальшої модифікації програми.

Налагодження програми можна починати до написання всіх блоків, вставивши замість відсутніх блоків які-небудь оператори, що імітують їх роботу і дають певні відповіді. Структуровану програму легше потім уточнювати, вдосконалювати і навіть переробляти.

У цей час найпопулярнішою технологією програмування є об'єктно-орієнтоване програмування (ООП). *Об'єктно-орієнтоване програмування* [object-oriented programming] — конструювання програм у вигляді ієрархічно впорядкованих класів об'єктів, що описують дані та операції над об'єктами, які можуть взаємодіяти з іншими об'єктами. Центральною ідеєю ООП є інкап-

суляція, тобто структурування програми на модулі особливого вигляду, що об'єднує дані і процедури їх обробки, причому внутрішні дані модуля не можуть бути оброблені тільки передбаченими для цього процедурами. У різних варіаціях ООП цей модуль називають або класом, або абстрактним типом даних, або ж кластером. Кожний такий клас має внутрішню частину, яка називається реалізацією, і зовнішню частину, яка називається інтерфейсом. Доступ до реалізації можливий лише через інтерфейс. Звичайно в інтерфейсі розрізнюють властивості, які синтаксично виглядають як змінні, і методи, які синтаксично виглядають як процедури або функції. Клас може мати методи, які називаються конструкторами та деструкторами, що дозволяють під час виконання програми динамічно породжувати й знищувати екземпляри класу. Екземпляри одного класу схожі між собою і наслідують методи класу, але мають різні значення властивостей. Класи та екземпляри класів називають об'єктами, звідки й виникає назва об'єктно-орієнтоване програмування.

ООП дозволяє програмісту об'єднувати в один блок дані та програмний код, що обробляє їх.

Використання структурного програмування для помірно складних програм дає позитивні результати, але воно виявляється неспроможним, коли програма досягне певної довжини. У результаті були розроблені принципи об'єктно-орієнтованого програмування. ООП акумулює кращі ідеї структурного програмування і поєднує їх з могутніми новими концепціями, які дозволяють оптимально організувати ваші програми. ООП дозволяє розкласти проблему на складові частини. Кожна складова стає самостійним об'єктом, що містить свої власні коди і дані, які відносяться до цього об'єкта. В цьому разі вся процедура спрощується, і програміст отримує можливість оперувати з набагато більшими за обсягом програмами.

ООП дозволяє точніше моделювати проблему, що існує в реальному світі, для розв'язання якої пишеться програма. Оскільки об'єкти — незалежні, відокремлені від іншої частини програми, блоки коду, їх простіше налагоджувати, змінювати і використовувати. Якщо об'єкти добре сконструйовані, то можна використати повторно набагато більшу частину програми, ніж у структурному програмуванні.

ТЕХНОЛОГІЯ ПРОГРАМУВАННЯ МОВОЮ С++

Розділ 4

ВСТУП ДО ПРОГРАМУВАННЯ МОВОЮ С++

4.1. ПОНЯТТЯ АЛГОРИТМІЧНОЇ МОВИ. ЇЇ ТИПОВІ КОМПОНЕНТИ

Спілкування між споживачами інформації здійснюється за допомогою певної мови. *Мова* — це сукупність способів для фіксації повідомлень та передавання їх від джерела інформації до споживача. Набір символів з заданими правилами утворення з цих символів конструкцій, за допомогою яких описується порядок виконання алгоритму, називається *алгоритмічною мовою*. Теоретичну основу мов програмування складають алгоритмічні мови. Звичайно при розробці мови програмування високого рівня спочатку створюється алгоритмічна мова з тією ж назвою. Алгоритмічна мова, призначена для опису алгоритмів розв'язання задач на ЕОМ, називається *мовою програмування*. Мови програмування умовно класифікуються таким чином: машинні, асемблерів, машинно-орієнтовані та автокоди, процедурно-орієнтовані та проблемно-орієнтовані, об'єктно-орієнтовані, візуальні. З моменту появи ЕОМ мови програмування пройшли великий шлях розвитку, який можливо подати у вигляді ієрархії рівнів (рис. 4.1).

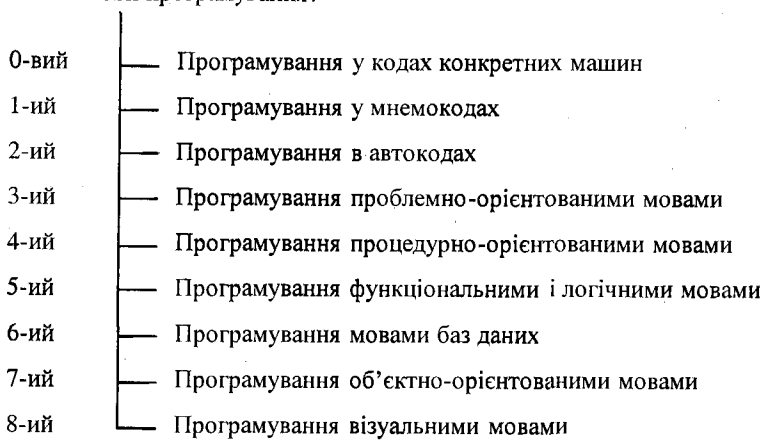


Рис. 4.1. Рівні автоматизації програмування

Нульовий рівень — програмування з використанням машинних мов [computer (machine) language]. *Машинна мова* — мова програмування, призначена для представлення програм і даних у формі, придатній для безпосереднього сприйняття їх пристроями даною обчислювальною машиною. Являє собою систему команд, даних і інструкцій, що мають форму двійкових кодів, які не вимагають трансляції і безпосередньо інтерпретуються процесором ЕОМ. Команда складається з коду операції, адрес операндів, ознак модифікації та індексації адреси. Код операції вказує дію, яку має виконати машина, наприклад, додавання, множення, ділення і т. д. Адреси операндів визначають дані, що беруть участь в операції. Ознака модифікації адреси складається з ознаки адресації, яка вказує дії над операндами. Ознака індексації вказує дії з індексними регістрами. Таке програмування досить складне в практичному здійсненні, важко здійснювати і перевірку таких кодів.

Перший рівень — програмування мовами символічного кодування (мнемокодами). *Мова символічного кодування* [symbolic language] — мова програмування, що орієнтована на конкретну ЕОМ і полягає в символьному кодуванні машинних операцій за допомогою певного набору мнемонічних символів. Мова утворюється внаслідок простої заміни в кодах машинних операцій тієї або тієї ЕОМ двійкових кодів операцій більш звичними для людини буквеними або буквено-цифровими кодами з десятковими цифрами. Двійкові коди адрес у командах замінюються десятковими кодами. Трансляція з такої мови називається мнемонікою і зводиться до простого перекодування. Використання такої простої вхідної мови набагато спрощує програмування і зменшує кількість помилок. Мнемокоди є базою для створення довершеніших систем автоматизації програмування, а також вони мають самостійне значення як системи програмування для малих ЕОМ, для яких не можна створити транслятори з мов високого рівня внаслідок обмежених можливостей машини.

Подальша автоматизація програмування на рівні машинно-орієнтованих систем складається у використанні *автокодів* (макромов) [autocod (macro language)]. Вхідною мовою автокодів є мова рівня один до декількох, тобто одній автокодовій інструкції відповідає кілька машинних команд. Основою автокодової мови є макрокоманди, що вказують функцію або процедуру за допомогою одного запису. Макрокоманди інтерпретуються в машинні команди спеціальними програмами. Структура команд автокоду визначається структурою команд і даними машинної мови, але автокод допускає на відміну від машинного застосування буквених позначень для операцій і адрес. У порівнянні з мнемокодами автокоди мають ряд переваг: наявність досконаліших методів виявлення помилок на етапі трансляції, різноманітних макрокоманд введення-виведення; зменшення трудових витрат на складання програм. Макромови нарівні з символічним кодуванням допускають використання команд, що не мають прямих аналогів у машинній мові, під час трансляції такі команди замінюються кількома машинними командами. Застосування макромови скорочує програму і значно полегшує процес програмування. До мови даного типу відноситься автокод «Інженер», мова Асемблера.

Мови першого й другого рівнів відносяться до машинно-орієнтованих мов. *Машинно-орієнтована мова* [computer-oriented language] — мова програмування, яка відображає структуру даної ЕОМ або даного класу ЕОМ.

Починаючи з четвертого рівня розташовані машинно-незалежні мови. *Машинно-незалежна мова* [machine-independent language] — мова програмування, структура та засоби якої не пов'язані ні з якою конкретною ЕОМ і дозволяють виконувати складені на ній програми на будь-яких ЕОМ, забезпечених трансляторами з цієї мови. *Проблемно-орієнтована мова* [problem-oriented language] — мова програмування, призначена для розв'язування певного класу задач (проблем). Мова по можливості використовує символіку та систему понять відповідної проблемної області. До цього типу мов відносяться Ліспі [Lisp], РПГ [RPG від Report

Program Generator], Симула. Ці мови використовуються для запису задач у термінології споживача. Алфавіт цих мов — терміни тих галузей науки і техніки, для яких складається програма. Ці мови не вимагають запису алгоритму як пов'язаної логічної послідовності дій. Досить визначити вхідні дані, вказати дії, які повинні проводитися над ними, і які результати потрібно отримати на виході. Всі інші функції покладаються на транслятор, що визначає, яка логічна схема потрібна для розв'язання задачі, і складає програму. Програма, написана проблемно-орієнтованою мовою, — компактна, зрозуміла фахівцеві в даній галузі знань. Для її складання не потрібно попередніх знань техніки програмування.

Процедурно-орієнтована мова [procedure-oriented language] — проблемно-орієнтована мова, що полегшує вираження процедури як точного алгоритму. Основні процедури, що зустрічаються в задачах конкретного класу, реалізовуються найпростіше у відповідній процедурно-орієнтованій мові. Кожна з них призначена для запису алгоритмів розв'язання задач певного класу. Виділяють три великих класи задач: наукові, інженерні та економічні. Специфіка кожного класу знайшла своє відображення в різноманітті мов програмування. Для розв'язання задач кожного з вказаних класів створювалися свої мови. Наприклад, мова Алгол призначена для розв'язання наукових задач, Фортран — для інженерних, Кобол — економічних, Снобол — задач обробки символьних даних.

У цих мовах обчислювальний процес записується як докладна послідовність певних процедур, що не залежать від машини. Використання процедурно-орієнтованих мов дозволило: спростити написання програм, скоротити час їх налагодження, виконувати програму, складену для однієї машини, на іншій.

Паралельно зі спеціалізацією мов у розрізі класів задач існують універсальні мови, які можна використати для програмування задач кількох класів. До таких мов належать мови ПЛ/1, Паскаль, С.

Мови п'ятого рівня призначені для програмування задач у галузі штучного інтелекту. Основною особливістю мов цього рівня, що відрізняють їх від усіх інших мов, є декларативний характер написаних на них програм. Декларативні мови дозволяють програмісту визначити правила за рішенням даної задачі, взаємозв'язку між об'єктами, з якими має справу програма. Декларативні мови поділяються на логічні та функціональні. *Мова логічного програмування* [rule-oriented language] — мова програмування, яка ґрунтується на принципі завдання сукупності правил без явної вказівки послідовності їх застосування. Найвідомішою

мовою логічного типу є Пролог. Будівельними блоками програми є множина об'єктів певної структури, а також функції і відношення, що зв'язують ці об'єкти. Програма мовою Пролог не є такою в традиційному розумінні, оскільки не містить керуючих конструкцій типу умовних операторів, операторів циклу або переходу. Вона являє собою модель певного фрагмента предметної області, про який ідеться в задачі, що розв'язується. Тому програмування Прологом вимагає іншого стилю мислення, відмови від поширених програмістських стереотипів. Замість того, щоб задати певну послідовність дій, що приводять до розв'язання задачі, в програмі мовою Пролог треба описати її вміст у термінах об'єктів і відношень між ними. Таким чином, замість алгоритму розв'язання задачі програміст складає його логічну специфікацію.

Мова функціонального програмування [functional language] — декларативна мова програмування, що ґрунтується на понятті функції. Функції в мові задають залежність, але не визначають порядок обчислень. З функціональних мов найвідомішою є Лісп.

Мови п'ятого рівня застосовуються для створення експертних систем, інтелектуальних інформаційних систем, інтелектуальних навчаючих систем.

До мов шостого рівня належать *мови баз даних* [database language]. Ця форма мови програмування і керування призначена для роботи з однією певною базою даних і здійснює ті дії програми, які мають бути автоматизовані.

Об'єктно-орієнтована мова — мова програмування, яка підтримує об'єктно-орієнтоване програмування. До даної групи мов належать C++, Smalltalk.

Всі мови ООП базуються на трьох основоположних концепціях — на інкапсуляції, поліморфізмі та успадкуванні. Інкапсуляція — це механізм, який об'єднує дані та код, що маніпулює з цими даними, захищає їх від зовнішнього втручання або неправильного використання. У ООП дані та програмний код, що обробляє їх, можуть бути об'єднані разом, тоді кажуть, що створюється об'єкт. Об'єкт — суть, яка включає не тільки дані, а й процедури (методи) їх обробки. Суттю може бути, наприклад, запис про студента. Об'єкт включає в себе всі дані, які необхідні, щоб описати суть і функції, або методи, які маніпулюють цими даними. Всередині об'єкта коди і дані можуть бути закритими для цього об'єкта або відкритими. Закриті коди або дані недоступні для тих частин програми, які існують поза об'єктом. Якщо коди і дані є відкритими, то незважаючи на те, що вони задані всередині об'єкта, вони доступні для інших частин програми.

Можливість описувати необхідні типи даних, використовуючи класи, дозволяє багато разів використати написану і налагоджену програму. Якщо вам потрібен майже такий самий об'єкт, як розроблений, але який має свої власні визначені характеристики, то механізм успадкування дає можливість більш легкого багаторазового використання об'єкта, після невеликого корегування. Успадкування — це процес, за допомогою якого один об'єкт може набувати (успадковувати) основних властивостей іншого об'єкта і додавати до них риси, характерні тільки для нього.

Поліморфізм дає можливість поводитися з об'єктами різного типу так, ніби вони є об'єктами одного типу. Поліморфізм використовує базовий клас як загальний тип для обробки множини похідних типів. У загальному значенні концепцією поліморфізму є ідея «один інтерфейс, множина методів». Це означає, що можна створити загальний інтерфейс для групи близьких за змістом дій. Поліморфізм допомагає знижувати складність програми, дозволяючи використання того самого інтерфейсу для завдання єдиного класу дій. Вибір же конкретної дії, в залежності від ситуації, покладається на компілятор. Поліморфізм дає змогу маніпулювати об'єктами різної міри складності шляхом створення загального для них стандартного інтерфейсу для реалізації схожих дій.

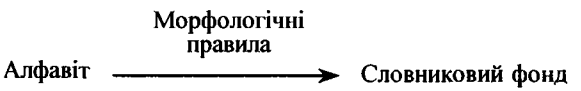
Поширення графічних інтерфейсів користувачів привело до появи візуальних середовищ розробки. *Візуальна мова* програмування [visual programming language] — мова взаємодії користувача з системою програмування, що реалізовується діалоговими засобами графічного інтерфейсу користувача. Поява візуальних мов була наслідком складності програмування графічного інтерфейсу програм, призначених для виконання в операційних середовищах типу Windows. Щоб полегшити програмування елементів графічного інтерфейсу і використовуються візуальні мови. Наприклад, малювання діалогового вікна введення даних зводиться до малювання його на екрані або до вибору і корегування прототипу, що пропонується в режимі діалогу. При цьому застосовуються покажчик миші, кнопки, меню та інші елементи. Візуальними мовами програмування можна розробляти як оболонки до програм, що вже існують, так і створювати нові. Для цього в мовах візуального програмування розроблений зручний механізм написання і підключення на базовій мові програмування високого рівня, такого як Бейсік, Паскаль, C++. До цієї групи мов належать Visual Basic, PowerBuilder, Borland C++ Builder.

Наведена класифікація є умовною, оскільки практично використовувані мови поєднують властивості різних мов з наведеної класифікації.

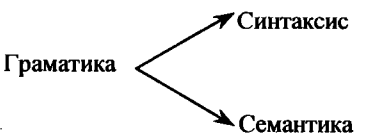
Кожна алгоритмічна мова має свою структуру. Структура мови визначається множиною початкових символів, з яких за допомогою сукупності правил утворюються мовні конструкції. Звичайно в мові виділяють: основні символи (алфавіт), слова, вирази, речення (оператори).

У спрощеному вигляді машинно-незалежна мова може розглядатися як набір слів, об'єднаних у речення за допомогою використання граматичних правил.

Для побудови слів існують певні правила. Набір правил для утворення слів описується морфологією мови. Словниковий склад визначає її лексику. Елементом будь-якої мови є алфавіт.



Мовна граматика складається з двох розділів: синтаксичного (задає правила і способи побудови словосполучень, різні типи речень та умови їх використання); семантичного (вивчає смислове значення слів та виразів).



Синтаксис мови описується в основному з допомогою метамов, основу яких складають металінгвістичні формули. Метамова [metalanguage] — мова, що використовується для опису інших мов. Металінгвістична формула — це пара:

$$< A >::= X ,$$

де A — назва мовної конструкції, що визначається, правила запису якої задаються правою частиною формули, для позначення назв конструкцій використовуються слова, що відображають

значення цих синтаксичних конструкцій; X — сукупність можливих символів алфавіту мови і назв конструкцій; знак «::=» означає «за визначенням є». Назви конструкцій у металінгвістичних формулах беруться в кутові дужки <>. Символ «¦» читається як « або». Наприклад,

<цифра>: : = 0¦1¦2¦3¦4¦5¦6¦7¦8¦9.

У лівій і правій частинах формул можуть стояти одні й ті самі найменування елементів, що визначаються, такі формули називаються рекурсивними. Наприклад,

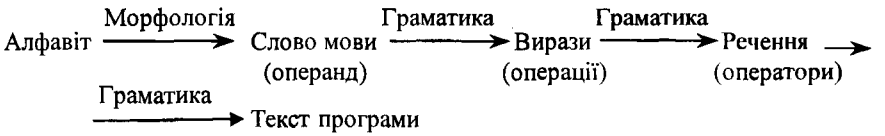
<ціле число без знаку>: : = <цифра> ¦ <ціле число без знаку> ¦ <цифра>.

Ця форма запису конструкцій мови називається нотацією Бэкуса-Наура.

Теоретична формула мови має вигляд:

Алгоритмічна мова = Словниковий фонд (алфавіт, морфологія) + Граматика (синтаксис, семантика).

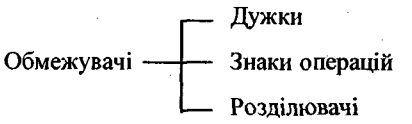
Ось схема побудови конструкції мови:



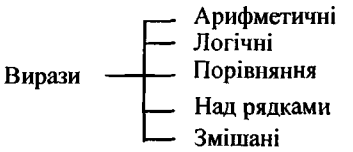
Вивчення будь-якої мови починається з алфавіту. Алфавіт є впорядкована певним чином сукупність різних знаків. Модель алфавіту може бути подана таким чином:

Слово являє собою поєднання символів, що мають певне значення. Слова є мінімальними одиницями мови, які мають власне смислове значення. Модель слова:

Окрім слів для утворення фраз і речень використовують набір обмежувачів — спеціальних символів, що задають форму мовних конструкцій.



Вираз в алгоритмічних мовах — послідовність змінних та/або констант, поєднаних знаками операцій та дужками. Види виразів:



Усі перераховані конструкції є окремими елементами мови, які не здатні описати закінчену думку. Для опису закінченої думки використовуються речення і текст, тобто група, набір взаємопов'язаних речень, що дозволяють передавати визначений зміст. Речення задає опис певної частини обчислювального процесу. Весь обчислювальний процес описується за допомогою ряду операторів. Їх поєднання є записом алгоритму.

4.2. ХАРАКТЕРИСТИКА МОВИ С++

Серед сучасних мов програмування мова С++ є однією з найпоширеніших.

С++ була розроблена співробітником науково-дослідного центру AT&T Bell Laboratories Бьярном Страуструпом 1979 року. Перед Страуструпом стояли два завдання: зробити С++ сумісною з мовою С, розширити С конструкціями об'єктно-орієнтованого програмування. Первинна назва «С з класами» була змінена 1983 року на С++, що відобразило походження нової мови від мови С.

Мова С розроблена Брайаном Керніганом і Деннісом Рітчи 1972 року під час роботи над операційною системою UNIX як універсальна мова загального призначення, що дозволяє ефективно реалізувати на ЕОМ задачі практично необмеженого спектра. Засоби мови дозволяють писати компактні програми, які за продуктивністю, ефективністю, затратами на компіляцію, компактністю коду і швидкодією програм можуть конкурувати з програмами, написаними мовою Асемблера. При цьому програми, написані С, є компактнішими і простішими в супроводі.

Мова С має ряд істотних особливостей, які виділяють її серед інших мов програмування. З одного боку, мова С підтримує повний набір конструкцій структурного програмування, модульність, блокову структуру програми, роздільну компіляцію. З іншого боку, в неї включені засоби програмування на рівні Асемблера (використання вказівників, побітові операції, операції зсуву). Тобто мова С має два рівня: нижній рівень — для системного програмувача; верхній рівень — для прикладного програміста.

Мова С — мобільна мова, що означає можливість перенесення програм з однієї обчислювальної системи в іншу практично без зміни тексту. Це досягається шляхом перенесення найбільш залежних від системи ЕОМ засобів, а саме засобів введення-виведення, розподілу пам'яті, маніпулювання з екраном із апаратної частини мови у бібліотеку стандартних функцій, які можуть доповнюватися специфічними функціями, що якнайконкретніше використовують особливості конкретної ЕОМ.

Мовні засоби С утримують повний набір операторів, які підтримують технологію структурного програмування, мовні засоби С++ — об'єктно-орієнтованого програмування.

Мова має широкий набір операцій, більшість з яких відповідає машинним командам і тому допускає пряму трансляцію. С підтримує роботу з вказівниками на об'єкти програми, які являють собою машинну адресу ОЗП даного об'єкта. Маніпуляція з адресами дозволяє ефективно використати ресурси пам'яті.

Мова С утримує у своєму складі препроцесор, який дозволяє вставляти тексти в початковий модуль до етапу трансляції.

Мова С має широкий спектр стандартних типів даних і зручний механізм композиції стандартних у складніші. Мова дає програмістові широку свободу у виборі синтаксичних конструкцій, допускає використання вказівників. У мові С є засоби роботи з системними і призначеними для користувача бібліотеками мови С.

Незважаючи на ефективність і потужність конструкцій мови С, вона відносно мала за обсягом. У ньому відсутні вбудовані оператори для виконання введення-виведення, динамічного розподілу пам'яті, керування процесами і т. ін. Вони реалізовані у бібліотеці стандартних функцій. Винесення цих функцій у бібліотеку дозволяє відділити особливості архітектури конкретного комп'ютера і угод операційної системи від реалізації мови, зробити програму максимально незалежною від деталей реалізації операційного середовища.

Подальшим етапом розвитку мови С стала мова С++, яка є супермножиною мови С. С++ підтримує об'єктно-орієнтоване програмування, що використовує класи, які є розвитком концепції структур, що дозволяє інкапсулювати оголошення даних і коду. С++ підтримує концепцію успадкування, коли один клас є похідним від іншого. С++ підтримує такі додаткові засоби, як шаблони та об'єктно-орієнтовану обробку виняткових ситуацій. С++ надає стандартну бібліотеку шаблонів разом з рядковим шаблоном. Завдяки цим можливостям можна писати програми, сумісні з іншими компіляторами ANSI С++. С++ розширює С-подібні частини мови, одночасно сприяючи використанню нових прийомів і стилів програмування і роблячи мову С++ значно багатшою за своїми можливостями від більшості мов високого рівня.

Мова С++ вже стала універсальною мовою для програмістів усього світу. Дана мова може бути застосована для програмування будь-якої задачі. Мова С++ часто використовується для таких проектів, як створення операційних систем, редакторів, систем керування баз даних, мультимедіа-програм, персональних інформаційних систем для розв'язання економічних, наукових, інженерних задач. Більша частина сучасного системного та прикладного програмного забезпечення розробляється саме цією мовою.

4.3. ОРГАНІЗАЦІЯ ІНТЕГРОВАНОГО СЕРЕДОВИЩА МОВИ С++ НА ПЕОМ

Необхідність переведення тексту програми, написаного алгоритмічною мовою, в мову машинних кодів конкретної ЕОМ визначає наявність в операційній системі спеціальних програмних засобів, що виконують вказані функції, які, власне, і є *середовищем,* або ж *інтегрованим оточенням мови.*

Головне місце в середовищі будь-якої мови належить комплексу програмних засобів, який називається транслятором. Існують три види трансляторів: компілятори, інтерпретатори, асемблери.

Компілятором називається системна програма, що на основі аналізу програми, написаної мовою високого рівня, генерує програму мовою машинних команд, яка є точною копією початкової програми з погляду виконуваних функцій. Програма, що надходить на вхід компілятора, називається *початковою програмою* або початковим модулем; програма, отримана на виході компілятора — *об'єктним модулем.* Тобто компілятор — це транслятор, що виконує перетворення програми, складеної початковою мо-

вою, в об'єктний модуль. У процесі трансляції кожен оператор програми замінюється послідовністю виконавчих машинних команд (або команд резервування пам'яті).

Хоч об'єктний модуль містить текст у машинних кодах, він не готовий до виконання, оскільки у ньому не реалізовані звертання з поточної програми до інших програм, що практично завжди є присутні, і не визначені конкретні адреси завантаження програми в ОЗП. Тому наступним після компіляції етапом обробки програми є етап редагування зв'язків, який виконує спеціальна програма, що називається редактором зв'язків.

Після компілювання здійснюється етап редагування зв'язку. На вхід редактора зв'язку надходить об'єктний модуль, після реалізації міжмодульних зв'язків на виході редактора формується завантажувальний модуль, тобто програма, яка готова до завантаження у пам'ять та виконання. Через те, що структура об'єктного модуля в межах одного типу ЕОМ однакова, виділення цього етапу обробки програми дає можливість об'єднати окремі частини програм, написаних різними мовами програмування.

Останнім етапом обробки програми є завантаження і виконання. Завантаження проводиться спеціальною програмою — завантажувачем. Ця програма спочатку встановлює адреси вільних полів пам'яті ОЗП, а потім проводить коригування адрес об'єктів програми та завантаження програми в ОЗП. Програма-завантажувач встановлює адреси вільних полів пам'яті ОЗП. Після цього керування передається першій команді програми.

Асемблером називають такий вид компілятора, який транслює оператори початкової програми у машинні команди за принципом «один в один». Асемблер здійснює підготовку програми машинною мовою шляхом заміни символічних імен операцій на машинні коди, а символічних адрес — на абсолютні номери, а також включення бібліотечних програм і генерацію послідовностей символічних команд шляхом зазначення конкретних параметрів у макрокомандах.

Інтерпретатор — вид транслятора, що здійснює пооператорну (покомандну) обробку і виконання програми чи запиту, на відміну від компілятора, що транслює всю програму без її виконання. Інтерпретатор транслює кожен оператор початкового модуля у певний проміжний код, інтерпретує його за допомогою однієї або кількох машинних команд і одразу виконує ці команди. На відміну від компілятора та асемблера інтерпретатор зазвичай не генерує об'єктний код, а видає лише результат роботи послідовного виконання операторів початкової програми.

Крім компонентів, що перелічені, до середовища алгоритмічної мови відносяться також бібліотеки стандартних функцій, тобто об'єктних модулів, які можуть викликатися з будь-якої програми, підготовленої програмістом за заздалегідь визначеними синтаксисом мови правилами. Це найчастіше математичні функції, функції введення-виведення і т. ін. До середовища алгоритмічної мови також належать спеціальні програмні засоби налагодження, які дозволяють виявити помилки, полегшити роботу з їх розпізнавання та усунення.

Кожна з перерахованих груп програмних засобів підтримки мови у визначених операційних системах може відрізнятися складністю виконання, масштабами виконаних функції, і практично всі з перерахованих засобів завжди є присутніми.

Розглядаючи процес доведення програми до робочого стану під час роботи з компілятором та інтерпретатором, можливо зробити висновок про складність роботи з компілятором та простоту процесу інтерпретації. Слід зазначити, що складність компіляції виявляється тільки в період налагодження програми. Налагоджений завантажувальний модуль може виконуватися за відсутності в ОЗП компілятора і редактора, що значно підвищує швидкість проведення розрахунків, заощаджує пам'ять та забезпечує вищу швидкість виконання. Робота з інтерпретатором є простішою в процесі налагодження програми, але виконувана програма зберігається в початковому вигляді й потребує присутності інтерпретатора, та, природно, швидкість її виконання невисока.

4.4. ТЕХНОЛОГІЯ НАЛАГОДЖЕННЯ ПРОГРАМ У СЕРЕДОВИЩІ СИСТЕМИ BORLAND C++

У теперішній час для ПЕОМ розроблено велику кількість реалізацій мови C++, авторами яких є різні фірми та організації. До цих реалізацій належать Borland C++, Microsoft Visual C++, Watcom C++, Symantec C++. Усі перераховані системи працюють за принципом компіляторів.

Фірма Borland є одним з провідних виробників С-компіляторів. Borland C++ версії 5 є наймогутнішим і таким, що має найбільші можливості компілятором на сьогоднішній день. Цій версії компілятора притаманна висока швидкість компіляції та ефективність коду, що створюється. Borland C++ складається з трьох компіляторів: компілятор С, компілятор C++, є підтримка Java, мови програмування для Internet.

Концепція Borland передбачає концентрацію повного набору програмних засобів, необхідних для обробки програм, написаних мовами високого рівня. До цих програмних засобів належать:

1) інтегроване середовище програмування;
2) компілятор початкового тексту програми;
3) редактор зв'язку (укладач);
4) бібліотеки заголовних файлів;
5) бібліотеки функцій;
6) програми утиліти.

Інтегроване середовище — програма, що має вбудований редактор текстів, підсистему роботи з файлами, систему допомоги (HELP), вбудований налагоджувач, підсистеми керування компіляцією та редагування зв'язків, компілятор та редактор зв'язку, менеджер проектів та інше.

Borland C++ має два варіанти роботи інтегрованого середовища:

bc.exe — для роботи в реальному режимі;

bcx.exe — для роботи в захищеному режимі.

Крім вбудованого в інтегроване середовище компілятора, пакет має ще один компілятор, що називається компілятором командного рядка (bcc.exe, bcc32.exe). **bcc.exe** — для компіляції 16-бітових DOS- і Windows-програм, написаних мовами C і C++. **bcc32.exe** — 32-бітовий компілятор. Транслює файли для використання в середовищі Windows NT або Win32s (під керуванням Windows NT, або Windows 95, або Windows 98). Компілятор командного рядка за замовчуванням після завершення компіляції автоматично викликає редактор зв'язку.

brcc.exe, brcc32.exe, brc.exe, brc32.exe і ***rlink.exe*** для компіляції описів ресурсів Windows і об'єднання результуючих двійкових файлів ресурсів у програмний файл з розширенням .exe.

make.exe (в захищеному режимі) і ***maker.exe*** (в реальному режимі) — для автоматизації створення багатофайлових програм;

tasm.exe — асемблер;

cpp.exe — препроцесор;

td.exe — налагоджувач;

tprof.exe — програма профілювання і зв'язані з нею файли;

файли конфігурації: ***turboc.cfg*** і ***tlink.cfg, bcc32.ctf*** і ***tlink32.cfg;***

tchelp.tch — довідка;

bcinst.exe — програма настроювання;

заголовні файли — ****.h***-файли;

бібліотечні файли — ****.lib*** і ****.obj***-файли.

Бібліотека стандартних функцій мови має широке використання і жодна програма користувача не зможе без них бути реалізованою. Кожна група функцій (за призначенням) має свій заголовний .h-файл, який повинен бути під'єднаний до програми користувача під час роботи з будь-якою функцією цієї групи. Під'єднання відбувається на рівні препроцесора мови.

Бібліотека стандартних функцій включає: функції класифікацій символів; функції перетворення рядка символів; функції маніпулювання рядками символів, математичні функції, функції керування процесами, які підтримують обробку помилок; функції динамічного розподілу пам'яті, функції, що підтримують роботу з датою та часом, функції введення-виведення та ін.

Налагодження програм може відбуватися за двома технологіями:

1) за допомогою автономного компілятора (за допомогою командного рядка);

2) за допомогою інтегрованого середовища розробки.

Автономні компілятори Borland C++ запускаються з командного рядка DOS.

bcc <options> <names>

де options — опції; names — ім'я модуля, який необхідно скомпілювати.

Якщо набрати на клавіатурі тільки bcc, то можна отримати абревіатуру опцій. Символи опцій залежать від регістра. Наприклад, опція — c — компілює в .obj-файл, але не компонує, а опція — C — дозволяє вкладені коментарі.

bcc — c mmain.cpp

Рядок містить завдання скомпілювати модуль mmain.cpp, в результаті буде створений об'єктний файл з розширенням .obj.

bcc mmain.obj ss.obj

Даний рядок викличе укладач для того, щоб об'єднати об'єктні модулі в файл, що виконується з розширенням .exe.

Відмінністю другої технології є те, що активізація програмних компонентів проводиться на різних етапах обробки програм не шляхом виклику за відповідними іменами, а шляхом активізації потрібної позиції меню інтегрованого середовища розробки.

Технологія налагодження однофайлових програм під керуванням Windows або DOS складається з таких кроків.

1. Запустіть IDE, викликавши в Windows програму Borland C++. (Рис. 4.1.)

2. Відкрийте новий файл початкового тексту: *File (Файл)/New (Новий)/Text edit (Текстовий файл).* Якщо початковий файл уже існує, відкрийте його, вибравши *File/Open/.*

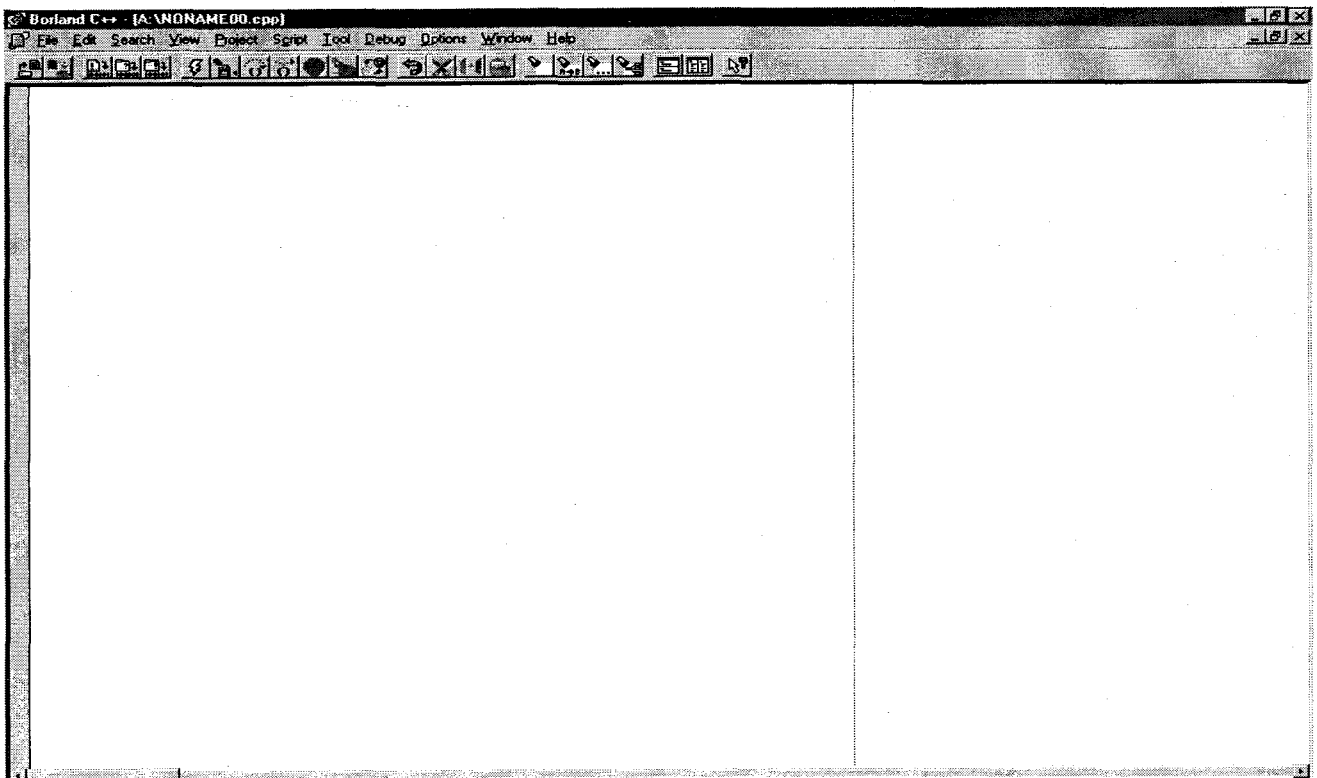


Рис. 4.1. Екран IDE Borland C++

3. Відкрийте оперативне меню, перемістивши курсор миші у вікно редактора і натиснувши один раз на праву кнопку миші, і виберіть команду *Target Expert (Експерт з мети).* На екрані з'явиться діалогове вікно *Target Expert.*

4. Якщо ви здійснюєте налагодження програми під керуванням DOS, то виберіть *Application (.exe) (додаток)* зі списку *Target Type (типи додатків)* і *DOS Standart (стандартна для DOS)* у списку *Platform (платформа), Small (мала)* в списку *Target Model (модель).*

Якщо ваша програма орієнтована на Windows, то виберіть тип результату *EasyWin[.exe]* зі списку *Target Type. Platform* треба встановити рівної *Windows 3.x [16]* і *Target Model* повинна бути *Large.* Ці установки діють за замовчуванням.

Виберіть кнопку *OK* для закриття діалогу *Target Expert.*

5. Введіть початковий текст програми і збережіть його, натиснувши клавішу *[F2],* з'явиться діалогове вікно, в якому необхідно дати ім'я вашій програмі. Ці ж дії можна виконати за допомогою команд меню.

Нові файли спочатку називаються NONAME00.CPP. При створенні кількох безіменних файлів у каталозі номер в імені автоматично збільшується від 00 до 99. Якщо ви зберігаєте файл за допомогою команд *File/Save* або *File/Save as...,* IDE запропонує ввести ім'я файла.

6. Скомпілюйте початковий файл (натисніть кнопку панелі інструментів *Compile* або виберіть команду *Project (Проект)/Compile(Компілювати).*

Якщо компілятор виявив помилки, то об'єктний код не формується і редактор зв'язків не викликається. Автоматично відкривається вікно повідомлень з посиланнями на помилки, внесіть виправлення в початковий текст. Повторіть компіляцію.

7. Скомпонуйте файл програми, вибравши команду *Link (Редактор зв'язків)* оперативного меню або натиснувши клавішу *[F9].* Ви можете виконати компіляцію і компонування за один крок, натиснувши кнопку панелі інструментів *Make all* або *Build all* або вибравши одну з команд *Project (Проект)/Make all (Переоформити)* або *Project/Build all(Сформувати)).* Для DOS-додатків виконання цього пункту обов'язкове.

8. Виконайте програму, натиснувши кнопку панелі інструментів *Run* або вибравши *Debug (Відладка)/Run (Виконати)* або натисніть *[Ctrl+F9].* Станеться компіляція і збирання програми. Якщо не виникне повідомлень про помилки, програма буде запущена.

Для DOS-додатків необхідно вийти в DOS і запустити виконуваний файл .exe.

Схема підготовки програми, що виконується, наведена на рис. 4.2.

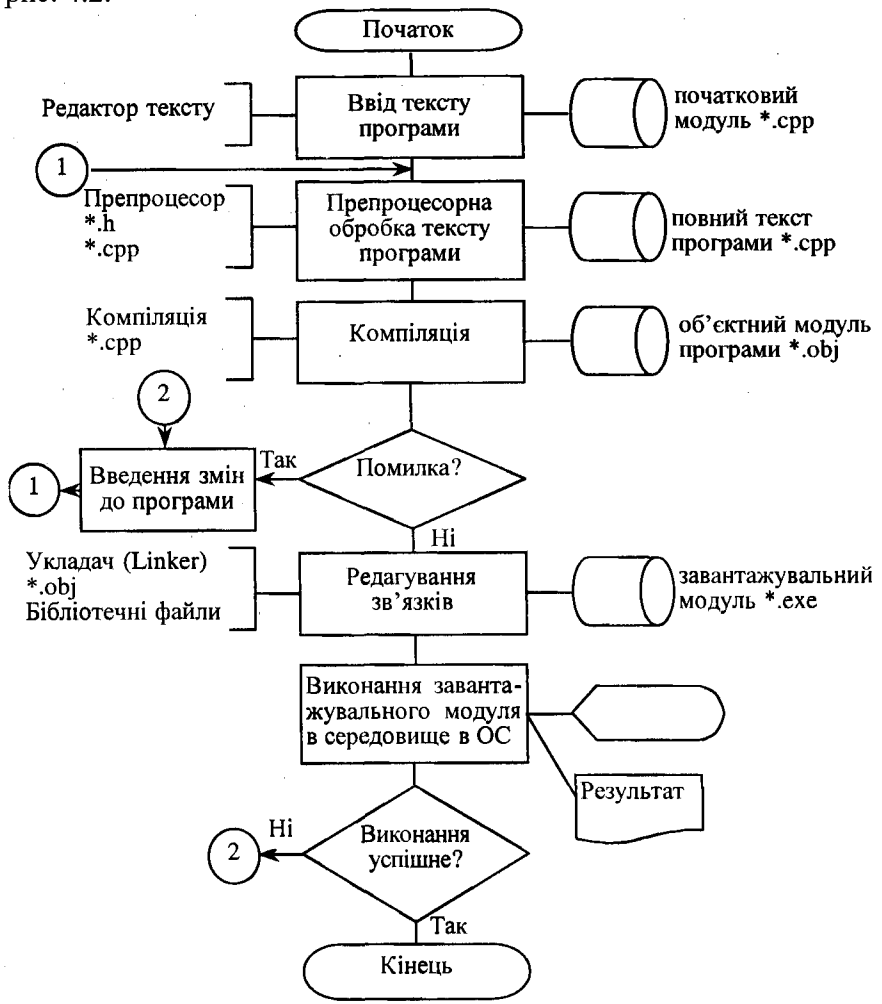


Рис. 4.2. Схема підготовки програми

Borland C++ може компілювати програми мовами C або C++. У загальному випадку, якщо програма має розширення .cpp, то вона компілюється як програма мовою C++, а якщо розширенням є C, то використовується компілятор C. Тому найпростіший спосіб примусити компілятор Borland C++ провести компіляцію за правилами C++ полягає у використанні розширення .cpp.

Розділ 5

ОСНОВНІ ТИПИ ДАНИХ

5.1. АЛФАВІТ, ІДЕНТИФІКАТОРИ, КЛЮЧОВІ СЛОВА, КОМЕНТАРІ

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

Алфавіт мови включає літери, цифри, спеціальні знаки.

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

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

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

Спеціальні знаки перелічені в табл. 5.1.

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

Пробільні символи — пробіл, символи табуляції, переведення рядка, переведення каретки, вертикальної табуляції. Ці символи відділяють один від одного лексеми, наприклад, константи та ідентифікатори. До пробільних символів належать коментарі.

Крім перерахованих знаків у мові використовуються керуючі послідовності (Esc-послідовності). *Керуючі послідовності* — спеціальні символи, що дозволяють представляти пробільні та нерафічні символи у символьних і рядкових константах. Їх застосовують для представлення символів або чисел, які не можна безпосередньо ввести з клавіатури. Для введення керуючих послідовностей, що дозволяють отримати візуальне представлення деяких символів, що не мають графічного аналога, використовується символ зворотної похилої риси (\). У табл. 5.2. перелічені допустимі керуючі послідовності Borland C++.

Таблиця 5.1

СПЕЦІАЛЬНІ ЗНАКИ

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

Таблиця 5.2

КЕРУЮЧІ ПОСЛІДОВНОСТІ МОВИ С++

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

Ідентифікатори — це умовні позначення змінних, функцій та позначок, інших об'єктів, що визначаються користувачем, використовуваних у програмі. Ідентифікатор може складатися з послідовності однієї чи кількох літер, цифр та знаків підкреслювання. Вони можуть бути практично будь-якими завдовжки. Ідентифікатори повинні розрізнюватися в перших 32 символах, число символів, що розрізнюються, може бути задане за допомогою пункту *Compile/Source команди Options/Project....* Ідентифікатор може включати прописні та рядкові букви, але вони не повинні збігатися за написанням з ключовими словами. Наприклад, Name, name, NAME — це абсолютно різні ідентифікатори.

Ідентифікатори змінних не повинні:

1) починатися з цифри;

2) містити пробіли;

3) містити спеціальні символи за винятком символу підкреслення _ ;

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

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

Нижче наведено кілька прикладів коректних і некоректних імен ідентифікаторів.

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

Ключові слова — це ідентифікатори, які мають певний зміст для компілятора мови С++. Вони зарезервовані в мові для спеціального використання. Ключові слова завжди пишуться малими буквами. У табл. 5.3 наведені ключові слова С++.

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

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

/*<послідовність символів>*/

Таблиця 5.3

КЛЮЧОВІ СЛОВА МОВИ BORLAND C++

__ asm	__ cdecl	__ cs	__ declspec	__ ds
__ es	__ except	__ export	__ far	__ fastcall
__ finally	__ huge	__ import	__ interrupt	__ loadds
__ near	__ pascal	__ rtti	__ saveregs	__ seg
__ ss	__ stdcall	__ thread	__ try	__ asm
__ cdecl	__ cs	__ ds	__ es	__ export
__ far	__ fastcall	__ huge	__ import	__ interrupt
__ loadds	__ near	__ pascal	__ saveregs	__ seg
__ ss	__ stdcall	asm	auto	bool
break	case	catch	cdecl	char
class	const	const_cast	continue	default
delete	do	double	dynamic_cast	else
enum	explicit	extern	false	far
float	for	friend	goto	huge
if	inline	int	interrupt	long
mutable	namespace	near	new	operator
pascal	private	protected	public	register
reinterpret_cast	return	short	signed	sizeof
static	static_cast	struct	switch	template
this	throw	true	try	typedef
typeid	typename	union	unsigned	using
virtual	void	volatile	wchar_t	while

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

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

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

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


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

Хоч мова С++ не допускає вкладеності коментарів будь-яких стилів, але компілятор Borland C++ дозволяє однорядковий коментар у стилі С++ вкладати всередину багаторядкового коментаря. Наприклад:

```
/* Це багаторядковий коментар,
усередину якого вкладений однорядковий коментар.
// Однорядковий коментар
// Ще один однорядковий коментар
Це закінчення багаторядкового коментаря */
```

Для цього необхідно включити опцію *Nested Comments* у вікні, яке з'являється при виконанні послідовності команд: *Options/Project/ + Compiler/Source*. Вкладені коментарі бувають корисні для налагодження окремих фрагментів програм, які містять численні коментарі.

5.2. ТИПИ ДАНИХ

5.2.1. Типи арифметичних констант та змінних

Тип даних характеризує велика кількість значень, які може приймати об'єкт; велика кількість операцій, які можуть застосовуватися до об'єкта; розмір пам'яті, що займається об'єктом. Основними елементами даних (об'єктами) мови є константи і змінні.

У С++ усі типи даних розбиваються на:

1) «порожній» тип (відсутність типу, тобто тип void);
2) скалярний тип;
3) тип «функція»;
4) агрегатований тип.

До скалярних відносять арифметичні типи, перелічувальний, вказівники, тип посилання. До агрегатованих типів відносять масиви, структури та об'єднання, а також класи. Типи так само можна розділити на основні та похідні. Основні типи — це void, char, int, float і double разом з short, long, signed і unsigned варіантами деяких з них. Похідні типи включають вказівники і посилання на інші типи, масиви, функції, класи, структури й об'єднання.

Константа — це об'єкт програми, який використовується для задавання постійних величин. Константа — фіксоване значення, яке не може змінюватися програмою. Константа — число, окремі символи, рядки символів.

У мові С++ розрізняють п'ять типів констант: цілі; з плаваючою крапкою; символьні; символьні рядки або рядкові, перелічувальний тип.

Ціла константа — десяткове, вісімкове, шістнадцяткове числа, що представляють ціле значення арифметичної константи.

Десяткова константа — це одна чи більше десяткових цифр від 0 до 9 зі знаком або без нього. Десяткова константа не повинна починатися з нуля, якщо це число не нуль. Десяткові константи можуть приймати значення від 0 до 429467295. Наприклад, 23, 989, –111.

Вісімкова константа — послідовність цифр від 0 до 7, яка обов'язково повинна починатися з нуля. Наприклад, 012, 076, 0111. Вісімкові додатні константи не повинні перевищувати 037777777777, для негативних констант абсолютні значення не повинні перевищувати 020000000000.

Шістнадцяткова константа — це послідовність цифр від 0 до 9 і букв від A до F, яка має обов'язково починатися з поєднання 0x або 0X. Наприклад, 0x12, 0xAF, 0x8B. Шістнадцяткові додатні константи не перевищують значення 0xFFFFFFFF, а для від'ємних констант абсолютні значення не повинні перевищувати 0x80000000.

Від'ємні константи виходять застосуванням операції унарний мінус до відповідної додатної константи.

Кожна ціла константа має тип, що визначає її представлення в пам'яті. Десяткові цілі константи можуть мати тип int (цілий), long int (довгий цілий), unsigned long int (беззнаковий довгий цілий).

Вісімкова і шістнадцяткова константи можуть мати тип int, unsigned int, long int, unsigned long int.

Якщо в записі константи зустрічається суфікс L(l), компілятор інтерпретує константу як long. Суфікс U(u) означає, що константа має тип unsigned. Дозволяється комбінувати обидва суфікси в будь-якому порядку. Наприклад, 12L, 765LU, 012l, 0x4LU.

У табл. 5.4 наведені діапазони значень констант різних типів для комп'ютера, на яких int має довжину 4 байта і тип long — довжину 8 байтів.

Якщо константа має суфікс L або l, її тип буде перший з long int, unsigned long int, який може відповідати її значенню.

Якщо константа має обидва суфікси U і L (ul, uL, UL і т. д.), її тип даних буде unsigned long int.

Таблиця 5.4

ЦІЛІ КОНСТАНТИ І ЇХ ТИП

Значення константи	Тип даних
Десяткові константи:	
від 0 до 32 767	int
від 32 767 до 2 147 483647	long
від 2 147 483 648 до 4 294 967 295	unsigned long
понад 4294967295	генерується помилка
Вісімкові константи:	
від 00 до 077777	int
від 0100000 до 0177777	unsigned int
від 02000000 до 017777777777	long
від 020000000000 до 037777777777	unsigned long
понад 037777777777	генерується помилка
Шістнадцяткові константи:	
від 0x0000 до 0x7FFF	int
від 0x8000 до 0xFFFF	unsigned int
від 0x10000 до 0x7FFFFFFF	long
від 0x80000000 до 0xFFFFFFFF	unsigned long
понад 0xFFFFFFFF	генерується помилка

Змінна — символьне позначення поля пам'яті, призначене для зберігання величини, яка може бути отримана та змінена програмою. До проведення операцій із змінними в тексті програми має бути наведення її оголошення за допомогою ключових слів, що визначають типи даних, які можуть бути записані у відведене для роботи поле. При оголошенні змінної їй може бути присвоєне початкове значення шляхом вміщування знаку рівності і константи після імені змінної, цей процес називається ініціалізацією і в загальному випадку має вигляд:

тип ім'я_змінної=константа;

Цілі змінні можуть бути: int, unsigned int, signed int, short int, unsigned short int, signed short int, int long, unsigned long int, signed long int.

Вони можуть бути описані без ініціалізації:

```
int a;
long c, d, f; unsigned long t;
```

і з ініціалізацією:

```
int a= –3;
unsigned int a=3;
long c, d, f=66L;
long m= –455;
```

Константа з плаваючою крапкою, тобто дійсна константа — це дійсне десяткове число, представлене в такій формі:

[<цифра>][.<цифра>][E|e][+|–][<цифра>];

<цифра> — одна чи більше десяткових цифр від 0 до 9;

<e> — ознака експоненти, що задається, символів E або e. В константі з плаваючою крапкою може бути опущена або ціла, або дробова частина константи, але не обидві відразу.

Експонента складається з символу експоненти, за яким іде цілочисельне значення експоненти, можливо, зі знаком плюс або мінус.

Наприклад:

15.75E1 15.75 0.15E2 15. E–2 –.17e2

При відсутності будь-яких суфіксів константи з плаваючою крапкою мають тип даних double. Можна присвоїти константі з плаваючою крапкою тип даних float, додавши до неї суфікс f або F. У такий же спосіб суфікс l або L присвоїть константі тип даних long double.

Константи з плаваючою крапкою можуть бути трьох типів:

float (плаваючий тип з одинарною точністю) займає в пам'яті 32 біта, з них один біт відводиться для знаку, 8 бітів для подання експоненти порядку, 23 біта для подання мантиси. Точність десяткових цифр 7. Діапазон порядку від –38 до +38;

double (плаваючий тип подвійної точності) займає в пам'яті 64 біта, біти пам'яті розподіляються таким чином: 1 біт для знаку, 11 бітів для експоненти і 52 біта для мантиси, точність десяткових цифр — 15, діапазон порядку від –308 до +308;

long double (довгий плаваючий тип подвійної точності) займає в пам'яті 80 бітів, з них 1 біт для знаку, 15 бітів для експоненти і 64 біта для мантиси, точності десяткових цифр — 19, діапазон порядку становить від –4932 до 4932.

Змінні з плаваючою крапкою описуються з використанням специфікаторів типу float, double, long double. Змінна з плаваючою крапкою повинна бути описана в програмі за допомогою вказаних вище типів. Наприклад, без ініціалізації:

```
float a, b;
double c, d, f;
long g;
```

з ініціалізацією:

```
float a=15.35;
double pi=3.14159265359;
long double=6.63e–34;
```


5.2.2. Типи символьних констант та змінних

Символьна константа — це літера, цифра, розділювач або ESC-символ, які взяті в поодинокі лапки. Символьна константа має таку форму:

'<символ>';

<символ> може бути будь-яким символом з множини зображуваних символів, у тому числі будь-яким спеціальним символом, за винятком таких символів: апостроф, зворотний слеш, новий рядок.

Наприклад, 'a', '! ', '\n'.

Для того, щоб використовувати як символ поодиноку лапку, потрібно записати у вигляді ESC-символа '\', для використання зворотної косої — '\\'.

Значення символьної константи є числове значення її внутрішнього коду. В С символьні константи вважаються даними типу int, вона зберігається в 16 бітах з нульовим старшим байтом. У С++ символьна константа має тип char.

Символьна змінна займає один байт пам'яті та описується з використанням ключового слова char. Наприклад,

```
char symbol='k'; char k;
```

Символьний рядок — це послідовність, що складається з одного чи більше одного символів, взятих у подвійні лапки. Інакше символьні рядки називають літерним рядком, рядковим літералом або рядковою константою. Символьний рядок має таку форму представлення:

"<символи>";

<символи> — це довільна кількість символів з великої кількості зображуваних символів, за винятком символів подвійна лапка, зворотний слеш, новий рядок. Всередині символьних рядків допускається використання Esc-послідовностей. Наприклад,

```
"Це рядкова константа"
"A" "$"
```

Для формування символьних рядків, що займають декілька рядків тексту програми, використовується комбінація символів — зворотний слеш і новий рядок. Зворотний слеш ігнорується компілятором, і наступний рядок вважається продовженням попереднього.

Наприклад, рядковий літерал

```
"Даний приклад показує, як можна\
автоматично\nздійснювати об'єднання\
рядків у дуже довгий рядок"
```

ідентичний літералу
"Даний приклад показує, як можна автоматично
здійснювати об'єднання рядків у дуже довгий рядок"

У мові С++ немає спеціального описувача для символьних рядків, вони можуть бути описані лише як масиви типу char, усі символи константи розташовуються в суміжних байтах пам'яті. Останнім елементом масиву завжди є Esc-символ нульовий символ ('\0'), який означає кінець рядка. Цей символ формується автоматично під час натиснення клавіші Enter. Тому реальний масив, який запам'ятовується, в пам'яті завжди міститиме на один елемент більше, ніж у зовнішньому представленні.

р	я	д	о	к	\0

Тип символьного рядка — масив елементів типу char[n+1], де n — це кількість одиничних символів, що об'єднуються в рядок, додатковий символ призначений для зберігання ознаки '\0'.

Наприклад,

```
char array[20];
char string[6]= "Слово";
```

Символьний рядок С і С++ має тип даних int.

5.2.3. Вказівник

У мові С++ передбачені можливості роботи з символічними адресами пам'яті, у всякий розташований об'єкт програми. Для цього вводитися поняття вказівника.

Константа типу вказівник визначає безпосереднє значення адреси будь-якої змінної, тобто оперативної пам'яті. Вказівник-змінна (або просто вказівник) — це змінна, призначена для зберігання адреси об'єкта певного типу. У мові С++ визначені дві спеціальні операції для доступу до змінних через вказівники: операція & і операція *. Результатом операції & є адреса об'єкта, до якого операція застосовується. Якщо ім'я змінної, використовуваної у програмі prt, то, додаючи перед ім'ям операцію визначення адреси &prt, отримаємо адресу цієї змінної пам'яті. Оператор y=&prt; присвоює адресу prt змінної y. Операція * — це операція звернення до змісту пам'яті за адресою, що зберігається в змінній вказівнику, або рівному вказівнику-константі. Наприклад, оператор x=*y; присвоює x значення змінною, записаною за адресою y. Якщо y=&prt; і x=*y; , то x=prt;.

Змінна типу вказівник може приймати у процесі виконання програми будь-які значення констант типу вказівник. Вона повинна бути обов'язково описана у такій формі:

<тип змінної> * <ім'я змінної>;

де тип — це будь-який допустимий тип (базовий тип вказівника), а ім'я — це ім'я змінної-вказівника. Базовий тип вказівника визначає тип змінної, на яку вказує вказівник.

Наприклад, int *p; змінна p являє собою вказівник на тип цілий тип.

char *t; оголошується вказівник з ім'ям t, який вказує на змінну типу char.

Можна змішувати оголошення вказівників і звичайних змінних в одному рядку. Наприклад,

int x, *y;

Оголошує x як змінна цілого типу, а y — як вказівник на цілочисельний тип.

5.2.4. Булевий тип даних

Тип даних bool призначений для булевих величин. Вони можуть набувати тільки двох значень — *true* (істина) і *false* (хибність). При використанні булевих типів у небулевих виразах вони автоматично перетворюються до типу цілих чисел. С++ продовжує повністю підтримувати фундаментальну концепцію про те, що ненульові цілі числа відповідають значенню істина, а нульове значення відповідає значенню хибність. Наприклад,

```
bool outcod;
outcod=false;
```

5.3. ТИПІЗОВАНІ КОНСТАНТИ, ТИМЧАСОВІ ЗМІННІ

Типізовані константи — це змінні, значення яких не можна змінити, вони задаються за допомогою модифікатора const перед типом. Наприклад,

```
const float pi=3,1415926;
const int maiint=32767;
```

Оголошення змінною з ключовим словом const створює змінну лише для читання. Така змінна має бути ініціалізована під час опису, і будь-яка спроба змінити її значення призведе до помилки компіляції.

const int i; // це невірне

Ім'я такої змінної не може знаходитись у лівій частині оператора присвоювання, а також бути аргументом операторів інкремента та декремент.

Модифікатор const використовується тільки зі змінними, що мають клас пам'яті static. Регістрові та автоматичні змінні ніколи не оголошуються константами.

Використання модифікатора const допомагає уникнути важковловимих помилок програмування. Наприклад, якщо в кількох місцях програми використовується одна й та сама константа з великою кількістю знаків, то є велика ймовірність зробити помилку під час набору її значення. У той же час помилка в наборі імені змінної, описаної з модифікатором const, буде негайно виявлена компілятором.

Модіфікатор volatile означає, що змінна може змінюватися не лише поточною програмою, а й іншими процесами, наприклад, програмою обробки переривань. Даний модифікатор повідомляє компілятору не робити якихось припущень відносно майбутніх значень змінної. Наприклад,

volatile int vint; // змінююче ціле

Використання модифікатора volatile дає можливість опису *тимчасових змінних,* тобто змінних, значення яких можуть присвоюватися неконтрольованим з боку програми способом, і навіть у тому разі, коли ці змінні описані з модифікатором const.

Наприклад,

volatile const vcint=100; //змінююча константа

У цьому випадку компілятор не допустить, щоб ваша програма змінила значення змінної, і в той же час не робитиме ніяких припущень про її вміст.

5.4. РОЗМІРИ ЗБЕРІГАННЯ ТА ДІАПАЗОН ЗНАЧЕНЬ ОСНОВНИХ ТИПІВ ДАНИХ

В C++ можна використати різні типи даних для представлення інформації, що обробляється і що зберігається. Дані кожного типу займають певну кількість байтів пам'яті й можуть приймати значення у відомому діапазоні. Розмір і допустимий діапазон для них у різних реалізаціях мови можуть відрізнятися. У табл. 5.5 подані всі допустимі комбінації базових типів даних, доступних для програм, орієнтованих на середовище DOS або 16-біт WINDOWS.

Таблиця 5.5

БАЗОВІ ТИПИ ДАНИХ ДЛЯ 16-БІТОВИХ СЛІВ

Тип даних	Ім'я оголошення	Розмір у байтах	Діапазон значень
Символьний	char	1	– 128 ÷ 127
Беззнаковий символьний	unsigned char	1	0 ÷ 255
Знаковий символьний	signed char	1	– 128 ÷ 127
Цілий	int	2	– 32768 ÷ 32767
Беззнаковий цілий	unsigned int	2	0 ÷ 65535
Знаковий цілий	signed int	2	– 32768 ÷ 32767
Короткий цілий	short int	2	– 32768 ÷ 32767
Беззнаковий короткий цілий	unsigned short int	2	0 ÷65535
Знаковий короткий цілий	unsigned short int	2	– 32768 ÷ 32767
Довгий цілий	long int	4	– 2 147 483 648 ÷ 2 147 483 647
Знаковий довгий цілий	signed long int	4	– 2 147 483 648÷ 2 147 483 647
Беззнаковий довгий цілий	unsigned long int	4	0 ÷ 4 294 967 295
З плаваючою крапкою	float	4	– 3.4E – 38 ÷ 3.4E + 38
З плаваючою крапкою подвійної точності	double	8	–1.7E – 308 ÷ 1.7E + 308
Довге з плаваючою крапкою подвійної точності	long double	10	– 3.4E – 4932 ÷ 3.4E + 4932

У табл. 5.6 міститься інформація про 32-бітові типи даних

Таблиця 5.6

БАЗОВІ ТИПИ ДАНИХ ДЛЯ 16-БІТОВИХ СЛІВ

Тип даних	Ім'я оголошення	Розмір у байтах	Діапазон значень
Символьний	char	1	–128 ÷ 127
Беззнаковий символьний	unsigned char	1	0 ÷ 255
Знаковий символьний	signed char	1	–128 ÷ 127
Цілий	int	4	–2147483648 ÷ 2147483647
Беззнаковий цілий	unsigned int	4	0 ÷ 4294967295
Знаковий цілий	signed int	4	–2147483648 ÷ 2147483647
Короткий цілий	short int	2	–32768 ÷ 32767
Беззнаковий короткий цілий	unsigned short int	2	0 ÷ 65535
Знаковий короткий цілий	signed short int	2	–32768 ÷ 32767
Довгий цілий	long int	4	–2 147 483 648 ÷ 2 147 483 647
Знаковий довгий цілий	signed long int	4	–2 147 483 648 ÷ 2 147 483 647
Беззнаковий довгий цілий	unsigned long int	4	0 ÷ 4 294 967 295
З плаваючою крапкою	float	4	–3.4E–38 ÷ 3.4E+38
З плаваючою крапкою подвійної точності	double	8	–1.7E–308 ÷ 1.7E+308
Довге з плаваючою крапкою подвійної точності	long double	10	–3.4E–4932 ÷ 3.4E+4932

Розділ 6

ВИРАЗИ ТА ОПЕРАЦІЇ. ТЕХНІКА ВИКОРИСТАННЯ В ПРОГРАМАХ СТАНДАРТНИХ ФУНКЦІЙ

6.1. ВИРАЗ

Вираз — послідовність змінних, функцій та констант, поєднаних знаками операцій та дужками. Вирази за типом виконуваних операцій поділяються на:

1) арифметичні операції;
2) порозрядні логічні операції;
3) операції зсуву;
4) логічні операції та операції відношення;
5) операцію умови;
6) операцію присвоювання;
7) визначення розміру в байтах (операцію sizeof);
8) перетворення типів у виразах;
9) операції зведення типу;
10) операції над вказівниками.

Вираз може складатися з однієї чи більше операцій і визначати виконання цілого ряду елементарних кроків за перетворення інформації. Елементарна операція з перетворення інформації задається знаком операції. Операнди — це змінні, константи або інші вирази.

За числом операндів, що беруть участь в операції, їх поділяють на:

- *унарні* — в операції бере участь один операнд;
- *бінарні* — в операції беруть участь два операнди;
- *тернарні операції* — три операнди.

Операція присвоєння може бути як унарною, так і бінарною.

Всі операції впорядковані за пріоритетом, який визначає порядок інтерпретації виразу. Він може змінюватися круглими дужками. Якщо всередині дужок або при відсутності дужок узагалі у виразі є кілька операцій одного пріоритету, компілятор враховує додатково порядок виконання операцій, наприклад, зліва направо або справа наліво. У табл. 6.1 наведений перелік усіх операцій мови C++, впорядкованих у порядку убування пріоритету (тонка риска відділяє операції з однаковим пріоритетом).

Таблиця 6.1

ПРІОРИТЕТ ОПЕРАЦІЙ BORLAND C++ І ПОРЯДОК ОБЧИСЛЕННЯ (АСОЦІАТИВНІСТЬ)

Назва	Символ операції	Порядок обчислення
Звернення до функції	()	Зліва направо
Виділення елемента масиву	[]	Зліва направо
Виділення поля структурною змінною	.	Зліва направо
Виділення поля структурною змінною за вказівником на її початок	–>	Зліва направо
Операція дозволу видимості	::	Зліва направо
Логічне заперечення (НІ)	!	Справа наліво
Порозрядне заперечення	~	Справа наліво
Унарний мінус	–	Справа наліво
Унарний плюс	+	Справа наліво
Інкремент	++	Справа наліво
Декремент	– –	Справа наліво
Операція отримання адреси операнда	&	Справа наліво
Операція звертання за адресою	*	Справа наліво
Перетворення типу	(тип)	Справа наліво
Операція визначення розміру в байтах	sizeof	Справа наліво
Виділення вільної ділянки в основній пам'яті	new	Справа наліво
Звільнення виділеної операцією new ділянки пам'яті	delete	Справа наліво
Пряме звернення до компонента класу за ім'ям об'єкта і вказівник на компонент	.*	Зліва направо
Непряме звернення до компонента класу через вказівник на об'єкт і вказівник на компонент	–>*	Зліва направо
Множення	*	Зліва направо
Ділення	/	Зліва направо
Ділення за модулем	%	Зліва направо
Додавання	+	Зліва направо
Віднімання	–	Зліва направо
Зсув ліворуч	<<	Зліва направо
Зсув праворуч	>>	Зліва направо
Менше, ніж	<	Зліва направо
Менше або дорівнює	<=	Зліва направо
Більше, ніж	>	Зліва направо
Більше або дорівнює	>=	Зліва направо

Закінчення табл. 6.1

Назва	Символ операції	Порядок обчислення
Дорівнює	==	Зліва направо
Не дорівнює	!=	Зліва направо
Порозрядне І	&	Зліва направо
Порозрядне виключне АБО	^	Зліва направо
Порозрядне АБО	|	Зліва направо
Логічне І	&&	Зліва направо
Логічне АБО	||	Зліва направо
Умовна операція	?:	Справа наліво
Присвоєння	=	Справа наліво
Присвоєння добутку	*=	Справа наліво
Присвоєння частки	/=	Справа наліво
Присвоєння остачі	%=	Справа наліво
Присвоєння суми	+=	Справа наліво
Присвоєння різниці	–=	Справа наліво
Присвоєння зсуву ліворуч	<<=	Справа наліво
Присвоєння зсуву праворуч	>>=	Справа наліво
Присвоєння І	&=	Справа наліво
Присвоєння виключне АБО	^=	Справа наліво
Присвоєння АБО	|=	Справа наліво
Операція «кома»	,	Зліва направо

6.2. АРИФМЕТИЧНІ ВИРАЗИ

У мові визначені такі арифметичні операції:

бінарні — додавання (+), віднімання (–), множення (*), ділення (/), ділення за модулем (%);

унарні — унарний мінус (–), інкремент (++), декремент (– –).

Операндами арифметичних виразів можуть бути арифметичні змінні, функції та арифметичні константи.

Адитивні операції. Операції додавання та віднімання належать до адитивних операцій. При адитивних операціях виконуються звичайні арифметичні перетворення за замовчуванням, тип результату є тип операндів після перетворення. У процесі виконання адитивних операцій ситуація переповнення або втрати

значності не контролюється. Якщо результат адитивної операції не може бути представлений типом операндів після перетворення, то інформація втрачається.

Операція додавання становить два операнди, операція віднімання віднімає другий операнд з першого. Один з операндів може бути вказівником, тоді інший повинен бути цілим значенням, результатом такої адресної операції буде значення вказівника плюс або мінус значення другого операнда, помножене на розмір об'єкта, на який посилається вказівник. Допускається віднімання двох вказівників, результат до знакового цілого типу шляхом ділення різниці на довжину типу, який адресується вказівниками. Результат представляє число елементів пам'яті даного типу між двома вказівниками.

Мультиплікативні операції. Операції множення, ділення і ділення за модулем належать до мультиплікативних операцій. Операнди в операціях множення і ділення можуть мати цілий або плаваючий тип, а в операції ділення за модулем лише цілий тип. Операція ділення над цілими числами виконується не так, як над даними з плаваючою крапкою. У першому випадку в результаті відкидається дробова частина, інакше проводиться операція відкидання, при цьому результат не округляється до найближчого цілого, а приводиться до меншого цілого числа.

Результатом операції ділення за модулем є остача від ділення першого операнда на другий.

В операціях додавання, віднімання, множення, ділення результат виконання операції визначається в тому ж вигляді, що й операнди після арифметичних перетворень. У результаті виконання операції ділення за модулем 2 відкидається дробова частина (операція відкидання). При цьому результат не округлюється до найближчого цілого, а приводиться до меншого цілого числа. Операція ділення за модулем 2 проводиться тільки з цілими числами.

Операція *інкремент* (++) збільшує значення свого операнда на одиницю. Є дві можливості виконання операції:

1. Символи ++ знаходяться зліва від змінної, ++a; (префіксна форма запису).
2. Символи ++ знаходяться справа від змінної, a++; (постфіксна форма запису).

Ці можливості відрізняються одна від одної тим, у який момент відбувається збільшення операндів:

якщо знак операції знаходиться ліворуч, то збільшення змінної відбувається перед тим, як вона виконується;

якщо знак операції знаходиться праворуч, то збільшення змінної відбувається після того, як її значення виконується.

Застосування цієї операції у вигляді ++a приводить спочатку до збільшення значення a, потім до виконання дії, в якому бере участь змінна a. Наприклад,

```
int a=1;
cout << ++a;
```

змінна a спочатку прийме значення 2, потім буде виведена на екран.

Операція a++ означає, що спочатку значення змінної a бере участь у виконанні відповідної дії, а потім збільшується. Наприклад,

```
int a=1;
cout << a++;
```

змінна a набуде значення 2, але заздалегідь буде виведене її колишнє значення 1, тобто приріст значення виконується після виконання оператора, в якому бере участь змінна.

Операція *декремент* (– –) зменшує значення свого операнда на одиницю. Техніка виконання декремента є аналогічною інкременту. Наприклад,

```
a– –; – –a; .
```

Операція *унарний мінус* (–) виконує заперечення свого операнда. Операнд повинен мати цілий або плаваючий тип. Під час виконання операції виконуються звичайні арифметичні перетворення.

```
// Арифметичні операції
//для цілих чисел
#include <stdio.h>
int main (void);
{
int a=10, b = 7,c=0;
c=a+b;
printf ("\na+b=%d",c);
c=a–b;
printf ("\na–b=%d",c);
c=a*b;
printf ("\na*b=%d",c);
c=a/b;
printf ("\na/b=%d",c);
c=a%b;
printf ("\na\x25");
printf ("b=%d",c);
c=a++;
printf ("\na++ %d",c);
printf ("\ta =%d",a);
c=++b;
```

```
// Арифметичні операції
//для чисел з плаваючою крапкою
#include <stdio.h>
int main (void);
{
float a=10, b=7,c=0;
c=a+b;
printf ("\na+b=%f",c);
c=a–b;
printf ("\na–b=%f",c);
c=a*b;
printf ("\na*b=%f",c);
c=a/b;
printf("\na/b=%f",c);
c=a++;
printf ("\na++ %f",c);
printf ("\ta=%f",a);
c=++b;
printf ("\n++b %f",c);
printf ("\tb=%f",b);
c=a– –;
```

```
printf ("\n++b %d",c);
printf ("\tb=%d",b);
c=a– –;
printf ("\na– – %d", c);
printf ("\ta=%d",a);
c= – –b;
printf ("\n– –b %d",c);
printf ("\tb=%d",b);
return 0;
}
```

```
printf ("\na– – %f", c);
printf ("\ta=%f",a);
c= – –b;
printf ("\n– –b %f",c);
printf ("\tb=%f",b);
return 0;
}
```

На екран буде виведений результат виконання програм:

```
a+b=17
a–b=3
a*b=70
a/b=1
a%b=3
a++ 10  a=11
++b 8   b=8
a– – 11   a=10
– –b 7   b=7
```

```
a+b=17.000000
a–b=3.000000
a*b=70.000000
a/b=1.428571
a++ 10.000000 a=11.000000
++b 8.000000 b=8.000000
a– – 11.000000 a=10.000000
– –b 7.000000 b=7.000000
```

Усі операції впорядковані за пріоритетом, який визначає порядок інтерпретації виразу. Порядок може бути змінений дужками. Якщо всередині дужок або за відсутності їх у виразі є кілька операцій одного пріоритету, то компілятор враховує додатковий порядок виконання операцій.

6.3. ПОРОЗРЯДНІ ЛОГІЧНІ ОПЕРАЦІЇ

C++ підтримує такі порозрядні логічні операції:

- порозрядне логічне І (&);
- порозрядне логічне АБО (|);
- порозрядне виключне АБО (^);
- порозрядне заперечення (~).

Операнди порозрядних операцій можуть бути будь-якого цілого типу. Над операндами виконуються перетворення за замовчуванням. Тип результату визначається типом операндів після перетворення.

У табл. 6.2 перелічені результати порозрядних операцій І, АБО, виключної АБО для всіх комбінацій двох однорозрядних операндів.

Таблиця 6.2

ПОРОЗРЯДНІ ЛОГІЧНІ ОПЕРАЦІЇ

Значення біта операнда 1	Значення біта операнда 2	a & b	a ^ b	a | b
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Оператор порозрядного заперечення є унарним оператором, який передує своєму операнду подібно до унарного мінуса, кожен біт з 0 встановлюється в 1 і кожен біт з 1 встановлюється в 0. Якщо операнд має знаковий біт, то біт знаку також інвертується. Порозрядна операція заперечення має найвищий пріоритет з усіх порозрядних операцій.

У табл. 6.3 наведені приклади використання порозрядних операцій для роботи з окремими бітами значень змінних.

Таблиця 6.3

ПОРОЗРЯДНІ ОПЕРАЦІЇ ДЛЯ РОБОТИ З ОКРЕМИМИ БІТАМИ ЗНАЧЕНЬ ЗМІННИХ

Операція	Опис	Приклад
&	Виконує операцію порозрядного логічного І. Порівнює кожен біт операнда ліворуч з відповідним бітом операнда праворуч. Якщо обидва, що порівнюються, біти дорівнюють 1, то біт результату встановлюється у 1, в іншому випадку — в 0.	a & b 1011 & 1010 дорівнює 1010
|	Виконує операцію порозрядного логічного АБО. Порівнює кожний біт операнда ліворуч з відповідним бітом операнда праворуч. Якщо один з бітів, що порівнюються, рівний 1, то біт результату встановлюється у 1, в іншому випадку — в 0.	a | b 1011 | 1010 дорівнює 1011
^	Виконує операцію порозрядного виключного АБО. Порівнює кожний біт операнда ліворуч з відповідним операндом праворуч. Якщо один з бітів, що порівнюються, дорівнює 1 (але не обидва відразу), то біт результату встановлюється у 1, в іншому випадку — в 0.	a ^ b 1011 ^ 1010 дорівнює 0001
~	Виконує операцію порозрядного заперечення. Якщо біт має нульове значення, встановлює його в 1, якщо 1 — в 0.	a~b 1011 дорівнює 0100

6.4. ОПЕРАЦІЇ ЗСУВУ

Операції зсуву зсувають операнд ліворуч (<<) або праворуч (>>) на число бітів, задане другим операндом. Обидва операнди повинні бути цілими величинами. Виконуються звичайні арифметичні перетворення. Тип результату — це тип лівого операнда після перетворення. Під час зсуву ліворуч праві біти, що звільняються, встановлюються в нуль. Під час зсуву праворуч метод заповнення лівих бітів, що звільняються, залежить, від типу, отриманого після перетворення першого операнда. Якщо тип unsigned, то вільні ліві біти встановлюються в нуль. В іншому разі вони заповнюються копією знакового біта. Результат операції зсуву не визначений, якщо другий операнд від'ємний. У табл. 6.4. наведені приклади використання операцій зсуву.

Таблиця 6.4

ОПЕРАЦІЇ ЗСУВУ

Операція	Опис	Приклад
>>	Зсув на один біт праворуч. Зсув праворуч операнда зліва від знаку операції на число двійкових розрядів праворуч від знаку операції.	a >> b 1011>>2 дорівнює 0011
<<	Зсув на один біт ліворуч. Зсув ліворуч операнда зліва від знаку операції на число двійкових розрядів праворуч від знаку операції.	a << b 1011<<2 дорівнює 1100

Зсув операндів ліворуч на 1, 2, 3 і більше за розряди — найшвидший спосіб множення на 2, 4, 8, і т. д. Запис a << b еквівалентний $a*2^b$. Зсув операндів праворуч на 1, 2, 3 і більше за розряди — найшвидший спосіб ділення на 2, 4, 8 і т. д. Запис a >> b еквівалентний $a / 2^b$.

Обмеженням операції зсуву є те, що число двійкових розрядів для зсуву операнда зліва може бути заданий лише константою або константним виразом, тобто виразом, що цілком складається з констант. Тому компілятор згенерує повідомлення про помилку в такому випадку:

```
int a = 4, b = 5, c;
c = a >> b;
```


6.5. ВИРАЗИ ВІДНОШЕННЯ

C++ підтримує такі *операції відношення (порівняння): менше* (<), більше (>), менше і дорівнює (<=), більше і дорівнює (>=), рівно (= =), не дорівнює (!=).

Операції відношення використовуються для опису умов ходу різноманітних алгоритмічних процесів, в основному для реалізації розгалуження та організації виходів з циклічних процесів. Операндами операцій відношення можуть бути константи, змінні цілого типу або з плаваючою крапкою, символьні дані, вказівники. Потрібно звернути увагу на використання цих операцій для даних з плаваючою крапкою. Найкраще для них працювати лише зі знаком < або >, оскільки внаслідок округлення результату знак рівності може не спрацьовувати.

Реалізація операцій відношення полягає у визначенні їх істинності. Якщо умова, задана операцією відношення, виконується, то формується значення «істина», якщо не виконується — «хибність». Машинною реалізацією «істини» в інтерпретації компілятора є будь-яка числова константа, що не дорівнює нулеві, «хибність» інтерпретується як нуль.

Операції відношення, що використовуються в логічних виразах, описані в табл. 6.5.

Таблиця 6.5

ОПЕРАЦІЇ ПОРІВНЯННЯ

Операція	Опис	Приклад
<	Менше. Повертає значення «істина», якщо результат обчислення виразу ліворуч менший, ніж виразу праворуч.	a < b 5 < 7 істина. 5 < 3 хибність. 5 < 5 хибність.
<=	Менше або дорівнює. Повертає значення «істина», якщо результат обчислення виразу ліворуч менший або дорівнює виразу праворуч.	a <= b 5 <= 7 істина. 5 <= 3 хибність. 5 <= 5 істина.
>	Більше. Повертає значення «істина», якщо вираз ліворуч більший, ніж вираз праворуч.	a > b 5 > 7 хибність. 5 > 3 істина. 5 > 5 хибність.
>=	Більше або дорівнює. Повертає значення «істина», якщо вираз ліворуч більший або дорівнює виразу праворуч.	a >= b 5 >= 7 хибність. 5 >= 3 істина. 5 >= 5 істина.
==	Дорівнює. Повертає значення «істина», якщо вираз ліворуч дорівнює виразу праворуч.	a == b 5 == 7 хибність. 5 == 5 істина.
!=	Не дорівнює. Повертає значення «істина», якщо вираз ліворуч не дорівнює виразу праворуч.	a != b 5 != 7 істина. 5 != 5 хибність.

Операція відношення дорівнює = = відрізняється від операції присвоєння =. Операція присвоєння =. присвоює змінній, що знаходиться зліва від знаку =, значення, що стоїть справа від =. Логічна операція = = перевіряє, чи є значення ліворуч від знаку таким самим, як і праворуч, але не змінює значень змінних.

У порівнянні з арифметичними операціями пріоритет операцій відношень вважається меншим, тобто запис x > y + 2 і x > (y + 2) вважається аналогічним. Операція відношення нижча пріоритетом за арифметичну операцію.

6.6. ЛОГІЧНІ ВИРАЗИ

Логічні операції призначені для роботи з виразами типу відношення, які необхідно об'єднати у складні вирази. Склад логічних виразів за пріоритетністю такий:

- ! — інверсія (Логічне «НІ»);
- && — кон'юнкція (Логічне «І»);
- || — диз'юнкція (Логічне «АБО»).

Опис логічних операцій представлений в табл. 6.6.

Таблиця 6.6

ЛОГІЧНІ ОПЕРАЦІЇ

Операція	Опис	Приклад
!	Логічне «НІ» Повертає значення «істина», якщо результат обчислення виразу праворуч від знаку хибний, в протилежному разі дає «хибність».	a!b !1 хибність. !0 істина.
&&	Логічне «І». Повертає значення «істина», якщо вирази ліворуч і праворуч від знаку істинні, в протилежному разі дає «хибність».	a&&b 5&&5 істина. 0&&5 хибність.
||	Логічне АБО. Повертає значення «істина», якщо вираз від знаку або ліворуч, або праворуч істинний, в протилежному разі дає «хибність».	a || b 1 || 0 істина. 1 || 1 істина. 0 || 0 хибність.

Логічні вирази обчислюються зліва направо, обчислення припиняються щойно буде встановлена істинність усього виразу. Логічні операції не виконують стандартні арифметичні перетворення. Вони оцінюють кожен операнд з погляду його еквівалентності нулеві. Порівняно з операціями відношення логічні операції мають нижчий пріоритет, тому аналогічними вважаються записи

```
a > b && b > c || b > d ;
((a > b) && (b > c)) || (b > d);
```


6.7. ОПЕРАЦІЇ З ВКАЗІВНИКАМИ

Вирази, що використовують вказівники, підпорядковуються тим самим правилам, що й звичайні вирази.

У мові C++ дозволено здійснення таких видів операцій над вказівниками:

1) присвоювання вказівників;
2) отримання адреси вказівника;
3) визначення значення поля пам'яті;
4) збільшення вказівника;
5) зменшення вказівника;
6) порівняння вказівників.

Присвоювання адреси вказівнику відбувається таким же чином, як і присвоювання значень під час ініціалізації змінної, розглянуте вище, тобто за допомогою однієї з операцій присвоювання. Можна присвоювати один вказівник іншому. Ця операція має значення лише в тому разі, якщо обидва вказівники адресують один і той самий тип. Наприклад,

```
#include <stdio.h>
int main (void)
{ int x;
  int *p1, *p2;
  p1=&x;
  p2=p1;
  printf("%p %p", p1,p2); // виводить адреси, що зберігаються в
                          // p1, p2, вони дорівнюватимуть return 0;
}
```

Отримання адреси вказівника відбувається, як було зазначено вище, за допомогою операції &, яка записується перед ім'ям змінної. Наприклад,

```
p = & n;
```

вміщує адресу змінної n в p. Операцію & можна розглядати як «взяття адреси». Тому попередній оператор можна прочитати як «p отримує адресу n».

Визначення значення поля пам'яті здійснюється за допомогою операції *, яка записується перед змінною типу вказівник, що визна-

чає адресу вказаного поля пам'яті. Тобто операція * повертає значення змінній, що знаходиться за вказаною адресою. Наприклад,

```
#include <stdio.h>
void main (void)
{ int n = 100, k; int *pl;
  p = & n; // p отримує адресу n
  k = * p; // k за допомогою p присвоює значення n
  printf ("%d", k); // виведення 100
}
```

p містить адресу змінної n, то k = * p; вмістити n в k. Операцію можна розглядати як «взяти значення за адресою», тому вищенаведений оператор можна прочитати як «k набуває значення за адресою p».

До вказівників можуть застосовуватися лише дві арифметичні операції: додавання й віднімання. Збільшення вказівника проводиться за допомогою звичної операції додавання або за допомогою операції інкремента ++. Зменшення вказівника аналогічно з попереднім проводиться за допомогою операції віднімання або операції декремент – –. Наприклад, якщо p — це вказівник на ціле, що містить значення 1000, то після виразу p++; вміст стане 1002, оскільки ціле має довжину 2 байти, то при збільшенні p на одиницю вказівник буде вказувати на наступне ціле. Вираз p+2 дає значення адреси 1004, оскільки 1000+2*2.

Щоразу коли вказівник збільшується, він указує на наступний елемент базового типу, щоразу коли зменшується — на попередній елемент. Можна додавати або віднімати з вказівників цілі числа.

```
p = p + 2;
```

Крім додавання та віднімання вказівників і цілих чисел допускається знаходження різниці двох вказівників. Ця операція використовується для знаходження довжини елемента, визначеного першим вказівником. Різниця між двома вказівниками перетворюється до знакового цілого значення шляхом ділення різниці на довжину типу, який адресується вказівниками. Результат представляє число елементів пам'яті даного типу між двома адресами.

Можливе порівняння двох вказівників. Порівняння двох вказівників однієї з операцій відношення має значення в тому разі, коли обидва вказівники адресують загальний для них об'єкт, наприклад, масив або рядок.

```
#include <iostream.h>
void main(void)
{ int a = 3, b = 3;
  int *p1, *p2;
```



```
  p1 = &a; p2 = &b;
  if (p1! = p2) // p1 і p2 не дорівнюються, тобто вони вказують
                // на різні об'єкти.
    cout<< "Вказівники p1 і p2 не дорівнюються."
  else
    cout << "Цей рядок не друкується."
}
```

Не можна множити чи ділити вказівники, не можна складати вказівники, не можна застосовувати операції зсуву до вказівників, не можна додавати або віднімати типи float або double.

6.8. ПЕРЕТВОРЕННЯ ТИПІВ ДАНИХ У ВИРАЗАХ

У виразах, написаних мовою C++, дозволяється використання різних типів операндів. Під час обчислення виразів деякі операції вимагають, щоб операнди мали відповідний тип, а якщо вимоги до типу не виконані, спричиняють примусове виконання потрібних перетворень.

У процесі написання виразів у програмах необхідно прагнути до використання змінних і констант одного типу, бо це дозволить виключити затрати часу на автоматичне перетворення даних, виключити помилки виконання програми.

Коли змінні та константи різних типів змішуються у виразах, то відбувається перетворення до одного типу. Компілятор перетворює всі операнди від нижчого типу до вищого. Послідовність типів від «нижчого» до «вищого» є такою:

char, unsigned char, short int, unsigned short int, int, unsigned int, long int, unsigned long int, float, double, long double.

Застосування unsigned підвищує ранг відповідного знакового типу.

Компілятор мови виконує перетворення типів даних відповідно до таких правил:

1. Якщо два операнди виразу мають різні типи, то перетворення відбувається до вищого з них. Цей процес називається підвищенням типу.

• Всі змінні типу char і short int перетворюються до типу int. Усі змінні типу float — до типу double.

• Якщо один з пари операндів має тип long, то інший операнд перетворюється до типу long. Якщо один з пари операндів має тип double, то інший операнд перетворюється до типу double. Якщо один з пари операндів має тип unsigned, то інший операнд перетворюється до типу unsigned.

2. У момент присвоювання обчисленого значення виразу змінній, що стоїть ліворуч від знаку присвоювання, відбувається автоматичне перетворення значення виразу до типу змінної. При цьому може відмічатися підвищення, що описано вище, та пониження типу, при якому величина приводиться до типу даних, що мають найнижчий пріоритет.

Для прикладу, розглянемо перетворення типів, показане на рис. 6.1.

```
char a;
int b;
float c; double d;
resultat = (a / b) + (c * d) — (c + b);
```

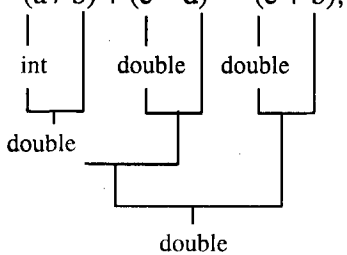


Рис. 6.1.

Підвищення типу звичайно не призводить до небажаних наслідків, оскільки розмір пам'яті даних вищого типу більший, ніж у перетворених значень. Невідповідність розмірів може призвести до ускладнень у процесі пониження типу. Наприклад,

```
long a = 12345678;
int b;
```

Що станеться, якщо b присвоїти значення a?

```
b = a;
```

Змінна b може запам'ятовувати значення –32768 до +32767, вона не зможе вмістити значення змінної a. Компілятор присвоїть b значення 24910. Це остача від ділення числа 12345678 на число 65536, тобто найбільше беззнакове ціле, яке може міститися в двох байтах.

Змішення цілих значень з плаваючою крапкою спричиняє перетворення типів, але результат не завжди можна передбачити.

```
int d = 4;
float f = 2.8;
d = d * f;
```

Чому дорівнюватиме d? У загальному випадку в змішаних виразах значення з меншим діапазоном розширяються до значень з

великим діапазоном, і щоб обробити вираз d * f , компілятор спочатку перетворює цілу змінну d в значення float, помножить це значення на f (4.0 * 2.8 = 11.2), а потім присвоїть результат зворотно змінної d. На останній стадії значення 11.2 звужується до типу int, присвоюючи зворотно змінної d значення 11.

Для виключення ситуації пониження типу у виразах можна використовувати операцію *зведення типів*. Вона передбачає запис виразу безпосередньо перед змінною, що потребує перетворення описувача типу, в який необхідно перетворити змінну. Описувач типу записується в круглих дужках і має такий формат:

(<тип>) <вираз>

де *<тип>* — це один зі стандартних типів даних або означених користувачем типів. Наприклад,

```
int a; float b;
a = (int)b + (int)1.5;
```

6.9. СТАНДАРТНІ ФУНКЦІЇ МОВИ І ЇХ ВИКОРИСТАННЯ У ВИРАЗАХ

Бібліотека стандартних функцій мови дуже широка і нараховує понад 300 функцій, що зберігаються у вигляді бібліотечних файлів у відкомпільованому вигляді (у вигляді об'єктних модулів *.obj у підкаталозі LIB) і підключаються до програми під час редагування зв'язків. Бібліотека — це зібрання функцій. У бібліотечному файлі зберігається назва кожної функції, об'єктний код функції та інформація, необхідна для редагування зв'язків. Коли програма робить посилання на функцію, що міститься в бібліотеці, укладач відшукує цю функцію і додає її код до програми.

Крім того, в бібліотеку входять заголовні файли, в яких містяться визначення необхідних типів для роботи з бібліотечними функціями та прототипи функцій.

У мовах C і C++ стандартна бібліотека сильніше інтегрована з мовою в порівнянні з іншими мовами програмування. Без використання функцій не може бути написана жодна C– або C++ програма.

Для використання стандартних функцій як операндів виразів необхідно виконати такі дії:

1. Встановити ім'я та формат звертання до функції, а також встановити ім'я заголовного файла, що включається, з підкаталогу INCLUDE, у якому визначені типи даних, що передаються, і констант, необхідних для використовування функції.

2. Перед функцією *main* програми користувача записати оператор препроцесора мови #include із зазначенням імені **.h**-файлу з заголовком функції:

#include <ім'я заголовного файлу>

3. У потрібному місці виразу записати звертання до функції за форматом:

ім'я функції (список фактичних параметрів);

Результат обчислень функції передається у пункт її виклику. Наприклад, необхідно обчислити $c = a^b$.

```
#include <iostream.h> // підключення бібліотеки класів введен-
                      // ня/виведення
#include <math.h>     // підключення бібліотеки математичних
                      // функцій
void main ()
{ float a, b, c;
  a = 4.0; b = 4.0;
  c = pow(a,b);       // обчислення піднесення a до ступеня b
  cout << c;          // виведення значення c
}
```

Склад стандартних функцій можна підрозділити на такі групи за їх призначенням:

1) Функції визначення класу символів та перетворення символів. Функції цієї групи виконують перевірку символів на приналежність до відповідного класу (алфавітно-цифрових символів, букв, керуючих символів, шістнадцяткових цифр, друкарських символів, малих букв, великих букв, пробільних символів, знаків пунктуації), а також перетворення букв з малих у великі й навпаки.

2) Функції перетворення даних. Функції цієї групи виконують перетворення числових даних з рядкового представлення в число типу int, long, float, double і навпаки.

3) Функції обробки рядка символів. Ці функції виконують порівняння рядків, пошук окремих символів або символів із заданого набору в рядку, копіювання рядків, перетворення символів у заголовні або прописні, заповнення рядка заданим символом, інвертування рядка (зміна на зворотну послідовність символів у рядку), з'єднання рядків, знаходження довжини рядка.

4) Функції керування каталогами. Ці функції дозволяють встановлювати шляхи доступу до файлів, створювати, змінювати, вилучати каталог, здійснювати пошук файла.

5) Функції керування файлами. Ці функції дозволяють отримати інформацію про файл, перевірити і змінити режим доступу і розмір файла, вилучити або перейменувати файл.

6) Функції введення-виведення. Функції цієї групи забезпечують введення-виведення потоком, низькорівневе введення-виведення, консольне введення-виведення та портів. Функції потоку забезпечують буферизацію вхідних і вихідних, форматоване та неформатоване введення-виведення. Функції консольного введення-виведення та введення-виведення портів є розширенням потокового введення-виведення для використання периферійних пристроїв. Низькорівневі функції введення-виведення працюють без буферизації та форматування, вони розглядаються як виклик до можливостей введення-виведення операційної системи.

7) Математичні функції. Ці функції забезпечують обчислення тригонометричних і гіперболічних функцій, знаходження цілої частини числа, абсолютного значення, кореня квадратного, піднесення до ступеня, обчислення логарифма, обчислення функції Бесселя.

8) Функції динамічного розподілу пам'яті. Ці функції дозволяють динамічно запитувати, перерозподіляти і звільняти пам'ять у різних моделях пам'яті.

9) Функції керування процесами. Термін «процес» відноситься до програми, яка виконується під керуванням операційної системи. Функції керування процесом дозволяють ідентифікувати процес унікальним номером, почати новий процес, завершити процес, обробити сигнал переривання, послати сигнал процесу, виконати функції під час завершення процесу, виконати команду операційної системи.

10) Функції, що забезпечують інтерфейс з DOS і BIOS. Ці функції забезпечують інтерфейс з BIOS за допомогою видачі відповідних переривань, а також інтерфейс з операційним середовищем DOS за допомогою системних викликів.

11) Графічні функції. Функції даної групи дозволяють керувати графічною системою, здійснювати установки параметрів зображення, отримувати зображення на екрані і параметри зображення.

12) Функції пошуку і сортування. Ці функції забезпечують бінарний і лінійний пошук і виконують швидке сортування.

13) Функції системного часу. У цю групу входять функції установки і читання системного таймера і перетворення дати і часу в різні представлення.

C++ підтримує всі функції введення-виведення C і визначає свою власну об'єктно-орієнтовану систему введення-виведення. До цієї бібліотеки відносяться бібліотека класів введення-виведення, класи потоків введення-виведення на базі масивів (форматування пам'яті).

Розділ 7

ЗАСОБИ ПРОГРАМУВАННЯ ЛІНІЙНИХ ТА РОЗГАЛУЖЕНИХ ПРОЦЕСІВ

7.1. ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ ВВЕДЕННЯ-ВИВЕДЕННЯ ПОТОКОМ

Важко собі уявити комп'ютерну програму, яка не має операторів введення-виведення. Як уже зазначалося вище, в мові С відсутні оператори введення-виведення, й операції введення-виведення здійснюються за допомогою стандартних функцій, що створюються розробниками компілятора мови. У бібліотеці є три групи функцій введення-виведення: потокозорієнтоване, низькорівневе, консольне введення-виведення.

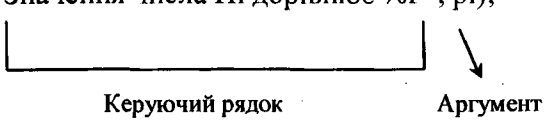
Найбільші зручності для користувачів створює група функцій, що здійснюють потокове введення-виведення, при якому дані передаються як безперервна послідовність символів. Цей вид передачі даних передбачає використання буфера, тобто спеціальної області ОЗП, яка призначена для об'єднання одиничних символів, що передаються в блоки. З буфера блоки передаються на зовнішній ЗП. Перевага використання буфера полягає у підвищенні швидкості обміну даними, у можливості коригування інформації, що передається.

Функції потокозорієнтованої передачі забезпечують форматоване введення-виведення з виконанням при необхідності перетворень. Наприклад, змінна в тексті програми може бути описана як float, а формат її виведення можна задати як int, тобто стандартні функції бібліотеки введення-виведення забезпечують можливість редагування даних під час виведення. Консольне введення-виведення відноситься до подій, що виникають від клавіатури або на екрані.

Найпоширенішими функціями при роботі з клавіатурою та екраном є консольні функції — *printf* та *scanf*, які визначені у заголовному файлі **<stdio.h>**. Функція *printf* здійснює форматоване виведення на екран. Формат функції *printf* є таким:

printf ("<керуючий рядок>",<список аргументів>);

printf("Значення числа Пі дорівнює %f" , pi);



Керуючий рядок (або форматний рядок, або рядок формату) функції *printf* описує вигляд розташування даних, поданих у списку аргументів при друкуванні. Список аргументів може містити одну або кілька змінних, констант або виразів, може бути й порожнім. Якщо у списку аргументів декілька об'єктів, то вони записуються через кому.

Керуючий рядок може містити пропуски, символи алфавіту, ESC-символи керування друком, а також специфікацію перетворень формату значень, що друкуються. *Рядок формату* — це символьний рядок, що складається просто з символів, які переносяться у відповідне місце без змін і специфікації перетворення.

керуючий рядок : := [пропуск][літерал][специфікація перетворення]

Специфікації перетворення формату мають такий вигляд:

%[прапори][ширина][.точність][модифікатор]<тип>

Кожна специфікація починається з символу процента, в квадратних дужках представлені необов'язкові елементи.

Прапори задають спосіб вирівнювання в полі виведення, наказують виведення знака у числа, визначають положення десяткової крапки, регулюють виключення хвостових нулів, задають вісімкові та шістнадцяткові префікси.

Ширина визначає мінімальне число символів для виведення, включаючи пробіли та нулі.

Точність визначає максимальне число символів для виведення, для змінних цілого типу визначає мінімальне число цифр під час виведення.

Модифікатор дозволяє вказати характеристики, пов'язані з розміром значення, що виводиться: N — near-вказівник, F — far-вказівник, h — значення типу short int, l — значення типу long, L — значення типу long double.

Необхідний тип даних вибирається відповідним символом перетворення.

Перелік символів типів, які можуть використовуватися у специфікації формату виводу, такий:

%c — поодинокий символ;

%s — рядок символів;

%d — десяткове число зі знаком;

%i — десяткове число зі знаком;

%e — число з плаваючою крапкою в експоненціальному (науковому) записі (e — мала літера);

%E — число з плаваючою крапкою в експоненціальному записі, (E — прописна літера);

%f — число з плаваючою крапкою в десятковому записі;

%g — за форматом %e чи %f, за якого виведення буде коротшим, ведучі нулі не виводяться;

%G — те саме, що і %g, але, якщо використовується експоненціальна форма запису, експонента пишеться прописними буквами — E;

%u — беззнакове десяткове число;

%o — беззнакове вісімкове число;

%x — беззнакове шістнадцяткове число, використовує малі літери;

%X — беззнакове шістнадцяткове число, використовує прописні літери;

%p — виводить значення вказівника.

Кожному аргументу із списку аргументів повинна обов'язково відповідати специфікація формату виведення керуючого рядка.

Символ прапора керує вирівнюванням виведення і друком знаку числа, пробілів, префіксів вісімкової і шістнадцяткової систем числення. У специфікації формату може бути задано не більше за один прапорець. У табл. 7.1. наведені символи прапорів і їхнє призначення.

Таблиця 7.1

ПРАПОРИ І ЇХНЄ ПРИЗНАЧЕННЯ

Прапори	Опис
–	Результат вирівнюється по лівому краю виділеного поля, що залишився праворуч, порожній простір, що залишився, заповнюється пробілами. За замовчуванням результат вирівнюється праворуч, заповнюється пробілами або нулями зліва.
+	Виводиться знак «+» або «–». За замовчуванням знак «–» з'являється тільки для від'ємних значень із знаком.
Пробіл	Перед додатними числовими значеннями виводиться пробілами, а перед від'ємними — знаком мінуса. Пробіл ігнорується, якщо одночасно вказані прапори «пробіл» і «+».
#	Виводиться ідентифікатор системи числення. У разі символа перетворення x або X ненульові аргументи випереджаються префіксом 0x або 0X відповідно. У разі символа перетворення o результат випереджається цифрою 0. Якщо використовуються e, E або f, результат завжди буде містити десяткову крапку, навіть якщо за точкою не потрібно ніяких цифр. Звичайно, десяткова крапка з'являється, тільки якщо за нею йде цифра. Якщо використовуються g або G, результат міститиме десяткову крапку, хвостові нулі не усуваються. Ігнорується, коли використовується з форматом c, d, i, u, s.

Значення ширини встановлює мінімальну ширину поля для виведення. Якщо кількість символів у значенні, що виводиться, менша, ніж задано в поле довжини, що виводиться, значення доповнюється пробілами до заданої мінімальної довжини. Якщо значення ширини задане з початковими нулями, нулі додаються до значення, що виводиться, до заданого мінімального розміру. Недостатня ширина поля ні в якому разі не приведе до відкидання поля. Якщо кількість символів результату більша, ніж оголошена ширина поля, поле буде просто розширене до необхідної ширини. Наприклад,

```
flloat t=15.54321;
printf ("\n%f", t);
printf("\n%10f", t);
printf ("\n%012f", t);
```

На екран буде виведений такий результат:

```
15.543210
 15.543210
00015.543210
```

Розглянемо ще один фрагмент програми:

```
printf ("\n/%d/", 123);
printf ("\n/%2d/", 123);
printf ("\n/%10d/", 123);
printf ("\n/%–10d", 123);
```

Результат виконання програми має такий вигляд:

```
/123/
/123/
/      123/
/123      /
```

Специфікація формату %d не містить модифікаторів, тому поле виведення має ширину, що дорівнює кількості цифр даного числа. Специфікація формату %2d вказує, що ширина поля повинна дорівнювати 2, але, оскільки число складається з трьох цифр, поле автоматично розширюється до необхідного розміру. Специфікація %10d показує, що ширина поля дорівнює 10, між символами / є сім пробілів і три цифри, число зсунуте до правого краю поля виведення. Специфікація %–10d вказує ширину поля, що дорівнює 10, а знак «–» приводить до зсуву всього числа до лівого краю.

Значення точності завжди починається з крапки (.), що відділяє її від попереднього значення ширини. За крапкою повинно йти ціле значення, воно визначає точне число символів, які мають бути надруковані, або місце десяткової крапки. Інтерпретація точності залежить від типу елемента, що форматується.

За замовчуванням точність рівна 1 для символів перетворення d, i, o, u, x, X; 6 для e, E, f; виводяться всі значущі цифри для символів перетворення g, G; виводиться символ для c; виводяться всі символи до нульового символа для S.

Для символів перетворення d, i, o, u, x, X визначає мінімальне число цифр, які будуть виведені. Якщо число цифр у результаті менше, ніж точність, значення, що виводиться, додається нулями. Значення не відсікається, якщо кількість цифр перевищує точність. Для символів перетворення e, E, f точність вказує число цифр, які будуть надруковані після десяткової крапки. Остання цифра, що виводиться, заокруглюється. Для символів g і G точність вказує максимальне число значущих цифр, які будуть виводитися. Для символів перетворення s визначає максимальне число символів, що виводяться. Символи, що перевищують точність, не друкуються.

Наприклад,

```
float pi=3.1415926
printf ("\nЧисло пі = %10.4f", pi);
```

Результат: Число пі = 3.1415

%10.4f виводить число з довжиною в десять цифр, причому під дробову частину відводяться чотири символи.

```
#include <stdio.h>
#define str "Символьний рядок"
 void main()
{ printf ("\n/%f/", 1234.98);
  printf ("\n/%e/", 1234.98);
  printf ("\n/%4.2f/", 1234.98);
  printf ("\n/%3.1f/", 1234.98);
  printf ("\n/%10.3f/", 1234.98);
  printf ("\n/%10.3e/\n", 1234.98);
  printf ("/%2s/\n", str);
  printf ("/%20.s/\n", str);
  printf ("/%20.6s/\n", str);
  printf ("/%–20.6s/\n", str);
}
```

Результат матиме такий вигляд:

```
/1234.980000/
/1.23498e+03/
/1234.98/
/1235.0/
/ 1234.980/
/ 1.23e+03/
/Символьний рядок/
/  Символьний рядок/
/           Символ/
/Символ           /
```


Специфікація формату %f вказує, що ширина поля виведення і точність беруться за замовчуванням, ширина дорівнює кількості цифр даного числа плюс символ на крапку, число цифр праворуч від десяткової крапки дорівнює 6. При використанні специфікації %e виводиться одна цифра зліва від десяткової крапки і шість праворуч. Тому в інших специфікаціях формату для даних з плаваючою крапкою задана точність, при цьому проводиться округлення даних, що виводяться. Необхідно звернути увагу на те, що специфікація точності обмежує число символів, що виводяться на друк. Специфікації формату %20.6s і %-20.6s вказують на необхідність виведення тільки шести.

Модифікатори F і H дають можливість користувачеві подавити угоди за замовчуванням про адресацію в моделі пам'яті, що використовується. F використовується для виведення значень, які були описані як far в малій моделі пам'яті. N призначений для значень near в середній, великій і надвеликій моделях пам'яті. Модифікатори F і T мають бути використані лише з символами перетворення "%s", "%p", оскільки вони мають значення тільки з аргументами, що задаються як вказівники. Модифікатори L, l, h визначають розмір очікуваного результату: h використовується з символами перетворення d, i, o, x і X для зазначення того, що тип аргументу short int, або з u — для зазначення, що тип аргументу short unsigned int. l використовується з символами перетворення d, i, o, x і X для зазначення того, що тип аргументу long int, з символом u — для зазначення, що тип аргументу long unsigned int, з e, E, f, g, G — для зазначення того, що тип аргументу double, а не float.

scanf ("<керуючий рядок>", <список аргументів>);

Функція scanf здійснює введення даних з клавіатури.

У керуючому рядку *scanf* використовуються ті самі специфікації формату, що й для функції printf. Потрібно зазначити, що специфікації перетворення %e, %E, %f, %g та %G еквівалентні значенням з плаваючою крапкою при введенні. Специфікація %s примушує функцію scanf читати символи, поки не зустрінеться спеціальний символ — пробіл, новий рядок, табуляція, вертикальна табуляція або переведення формату. Різниця між цими двома функціями полягає в особливостях списків аргументів. У списку аргументів *scanf* дозволяється використовувати тільки вказівники на відповідні значення (тобто перед ім'ям змінної має бути записаний &). Коли необхідно ввести значення рядковій змінній, використовувати символ % не треба. Після виконання функції *scanf* здійснюється автоматичне переведення курсора на наступний рядок.

Наприклад,

```
#include <stdio.h>
void main()
{ int ncel; float nplav; char nchar[20];
  printf ("\nВведіть будь-яке ціле число, не більше трьох розрядів");
  scanf ("%d", & ncel);
  printf ("\nВведіть будь-яке число, не більше трьох розрядів\n");
  printf ("в цілій і дробовій частинах");
  scanf ("%f", & nplav);
  printf ("\nВведіть ваше прізвище");
  scanf ("%s", nchar);
  printf ("\n%s", nchar);
  printf ("\Ви ввели два числа %3d %7.3f", ncel, nplav);
}
```

Як приклад наведемо ще одну програму, яка демонструє опис різних змінних, введення їх з клавіатури і виведення на екран дисплея.

```
#include <stdio.h>
void main()
{ int a;
  char b;
  float c;
  double d;
  short x;
  long y;
  unsigned z;
  scanf ("%d %c %lf %d %u", &a, &b, &c, &d, &x, &y, &z);
  printf ("%d %c %19.11f\n %ld %u", a, b, c, d, x, y, z);
}
```

Як і в С, введення і виведення в С++ не є частиною мови, а забезпечуються зовнішніми бібліотечними модулями.

7.2. ОПЕРАТОР ПРИСВОЮВАННЯ

Оператор присвоювання є основним оператором, що реалізовує переміщення даних у межах ОЗП.

Формат оператора присвоювання:

<змінна> = <вираз>;

Наприклад, x = 0; змінній x присвоюється значення 0. Можна присвоювати кільком змінним одні й ті самі значення шляхом використання численних присвоєнь в одному операторі, тобто x = y = z = 0; .

Якщо в процесі присвоювання типи змінних і результатів обчислення виразів не збігаються, то відбувається автоматичне перетворення типу результату до типу імені змінної. Якщо відбуватиметься перетворення їх цілого числа до символу, з довгого цілого — до цілого і з цілого — до короткого цілого, то старші біти втрачаються. 8 бітів втрачається при перетворенні 16-бітного цілого до символу, 16 бітів будуть втрачені при перетворенні від довгого цілого до цілого. Табл. 7.2. містить усі перетворення типів.

Таблиця 7.2

РЕЗУЛЬТАТИ ПЕРЕТВОРЕННЯ ТИПІВ

Початковий тип	Тип перетворення	Результат
unsigned char	signed char	Якщо значення >127, результат буде від'ємним числом
short int	char	Старші 8 бітів
int (16 біт)	char	Старші 8 бітів
int (32 біт)	char	Старші 24 бітів
int (16 біт)	short int	Немає
int (32 біт)	short int	Старші 16 бітів
long int	int (16 біт)	Старші 16 бітів
long int	int (32 біт)	Немає
double	float	Точність, результат округлюється
long double	double	Точність, результат округлюється

Для отримання перетворення, не вказаного в табл. 7.2, необхідно покроково перетворювати типи до отримання потрібного. Наприклад, для перетворення з double в int потрібно перетворити double в float, а потім float в int.

Крім простого присвоювання в мові С++ є складове присвоювання, яке проводиться за допомогою операцій інкремента і декремента, а також аддитивних операцій, що задаються знаками +, –, *, /, %, <<, >>, &, ^,¦. Формат даного оператора є таким:

<змінна><знак бінарної операції> = <вираз>;

На початку обчислюється значення виразу, потім виконується дія, вказана операцією (додавання, віднімання, множення і т.д. виразу і значення змінної), зі збереженнями попереднього значення змінної. Таким чином, запис a+ = 1 аналогічний запису a = a+1; в– = с; аналогічна у = в–с; . У табл. 7.3 наведені приклади використання складових операцій присвоєння.

Таблиця 7.3

СКЛАДОВІ ОПЕРАЦІЇ ПРИСВОЄННЯ

Операція	Опис	Приклад
+=	Присвоєння суми. Додати значення, що знаходиться праворуч від знаку, до значення змінної, розташованої зліва.	a += b; a += 5; додати 5 до a, тобто a = a + 5;
–=	Присвоєння різниці. Відняти значення, що знаходиться праворуч від знаку, до значення змінної.	a –= b; a –= 5; відняти 5 з a, тобто a = a – 5;
*=	Присвоєння добутку. Помножити значення, що знаходиться праворуч від знаку, до значення змінної.	a *= b; a *= 5; помножити 5 на a, тобто a = a * 5;
/=	Присвоєння частки. Розділити значення, що знаходиться праворуч від знаку, до значення змінної.	a /= b; a /= 5; розділити 5 на a, тобто a = a / 5;
%=	Присвоєння остачі. Зберегти в змінній, розташованій ліворуч від знаку, значення, рівне остачі від ділення цієї змінної на значення, що знаходиться праворуч.	a /= b; a %= 5; остача від ділення 5 на a присвоїти a, тобто a = a % 5;
<<=	Присвоєння лівого зсуву. Виконати зсув ліворуч значення змінної, розташованої ліворуч від знаку, на кількість бітів, вказаних праворуч.	a <<= b; a <<= 2; тобто a = a << 2;
>>=	Присвоєння правого зсуву. Виконати зсув праворуч значення змінної, розташованої ліворуч від знаку, на кількість бітів, вказаних праворуч.	a >>= b; a >>= 2; тобто a = a >> 2;
&=	Присвоєння порозрядного І. Виконати операцію порозрядного логічного І значення змінної, розташованої ліворуч від знаку, на значення, що знаходиться праворуч.	a &= b; a &= 2; тобто a = a & 2;
¦=	Присвоєння порозрядного АБО. Виконати операцію порозрядного логічного АБО значення змінної, розташованої ліворуч від знаку, на значення, що знаходиться праворуч.	a ¦= b; a ¦= 2; тобто a = a ¦ 2;
^=	Присвоєння порозрядного виключного АБО. Виконати операцію порозрядного виключного АБО значення змінної, розташованої ліворуч від знаку, на значення, що знаходиться праворуч	a ^= b; a ^= 2; тобто a = a ^ 2;

7.3. УМОВНА ОПЕРАЦІЯ

У мові С++ є одна тернарна операція — умовна операція ? : . Формат цієї операції є таким:

<змінна> = <вираз 1> ? <вираз 2> : <вираз 3>;

Ця операція складається з двох частин і містить три операнда. Виконання умовної операції починається з аналізу значення виразу 1. Якщо умова «істинна» (відмінна від 0), то обчислюється вираз 2. Якщо ця умова «хибна» (вираз дорівнює 0), то як значення всього умовного виразу обчислюється вираз 3. Завжди обчислюється тільки один з двох виразів, розділених двокрапкою, вираз 2 або вираз 3, але не обидва. Наприклад,

y = (a>b) ? a : b;

Даний оператор дозволяє присвоїти змінній *y* значення більшої змінної з *a* і *b*.

Операцію умови зручно використовувати тоді, коли є певна змінна, якій можна присвоювати одне з двох можливих значень.

7.4. УМОВНИЙ ОПЕРАТОР

Умовний оператор *if* використовується для організації розгалужених алгоритмів. Формат оператора є таким:

if (<вираз>)
  <оператор 1>;
[else <оператор 2>;]

де вираз у дужках — це простий або складний вираз типу відношення;

<оператор> — обов'язково простий або складений оператор. У свою чергу, може бути знову умовним. Складений оператор — група операторів, узятих у фігурні дужки. Оператор *else* не обов'язковий.

Стандартна форма оператора *if* зі складовим оператором є такою:

if (<вираз>)
{<послідовність операторів>;
}
[else
{<послідовність оператор>;
}]

Під час виконання оператора *if* спочатку обчислюється логічний вираз. Якщо результат «істина» (не дорівнює нулю), то виконується

оператор 1, записаний після виразу, якщо «хибність» — оператор 2, записаний за ключовим словом *else*. *Якщо* значення виразу «помилкове», але конструкція *else* відсутня, то керування передається оператору, наступному в програмі після оператора *if*. Якщо замість одного оператора після *if* або *else* потрібно виконувати кілька операторів, то вони вміщуються у фігурні дужки. Наприклад,

```
if (a > b && b > c)
  { S = S + b;
    S2 = S2 + c;
  }
else S1 = S1 + a;
```

Оператори 1 і 2 самі можуть бути операторами *if*, утворюючи так званий вкладений *if*. При вкладенні операторів *if* рекомендується для ясності групування операторів використовувати фігурні дужки, що обмежують <оператори 1> та <оператор 2>. Згідно з прийнятою в С++ угодою, слово *else* завжди відноситься до найближчого, що передує, йому *if*. Наприклад,

```
#include <stdio.h>
void main ()
{
 int a = 4, b = 9, c = 5;
 if (a > b)
 { if (b < c)
      c = b;
 }
 else c = a;
 printf ("\nc = %d", c);
 }
```

Результат виконання програми:

c=4

Якщо в цій програмі опустити фігурні дужки в операторі *if*, то програма матиме ось який вигляд:

```
#include <stdio.h>
void main ()
{
 int a = 2, b = 7, c = 3;
 if (a > b)
  if (b < c)
     c = b;
  else c = a;
 printf ("\nc = %d", c);
 }
```


У цьому разі ключове слово *else* відноситься до другого оператора *if*.

Результат виконання програми:

```
c=5
#include<stdio.h>
main()
{ int x;
  printf ("\nВведіть ціле число");
  scanf ("%d", & x);
  if (x%2 = = 0 && x>0)
    printf ("\nЧисло, яке введено, парне і додатне");
  else
    if (x%2! = 0 && x>0)
      printf ("\nЧисло, яке введено, непарне і додатне");
    else
      if (x%2 = = 0 && x<0)
        printf ("\nЧисло, яке введено, парне і від'ємне");
    else
        printf ("\nЧисло, яке введено, непарне і від'ємне");
}
```

Конструкція *if-else-if* може виконувати численні перевірки, але вона виходить довгою і не завжди зрозумілою. Іншим способом організації вибору з великої кількості варіантів є використання оператора множинного вибору *switch*.

7.5. ОПЕРАТОР МНОЖИННОГО ВИБОРУ

Часто в програмуванні виникає задача вибору одного з множини варіантів. Можна це зробити за допомогою вкладеного умовного оператора, але оператор множинного вибору робить текст програми наочним і лаконічним, дозволяє швидше реалізувати вибір.

Оператор має такий формат:

```
switch (<вираз>)
{ case <константа 1>:
  { <оператор 1>;
...
```

```
    [<оператор n>;]
    [break; ] }
  case <константа n>:
  { <оператор 1>;
...
    <оператор n>;
    [break; ]}
  [default: ]
  {[<оператор n>;]
  [break; ]}
}
```

Виконання оператора *switch* починається з обчислення виразу після ключового слова *switch*. Тип виразу може бути int, unsigned int, char, long int, unsigned long. Обчислене значення порівнюється зі значенням констант або константних виразів. У разі рівності виконуються оператори або оператор, що йдуть після ключового слова *case*. Потім керування передається на оператор, який стоїть після *switch*, якщо у знайденій групі *case* є оператор *break*, у протилежному разі виконуються наступні за ним оператори доти, поки в них не зустрінеться оператор *break* або не буде виконаний останній оператор у межах *switch*. Якщо не знайдено жодне значення константи, яке дорівнює виразу, то виконуються оператори, що йдуть після ключового слова *default*. Якщо в операторі *switch* немає групи *default*, то керування передається наступному оператору після оператора *switch*. Група *default* не обов'язкова і рекомендується її ставити останньою в списку варіантів. Наприклад,

```
// Демонстрація switch
#include<stdio.h>
#include<conio.h>
main()
{
  int y,a,b;
  char znak;
  printf ("\nВведіть a і b\n");
  scanf ("%d%d", &a,&b);
  printf ("\nВведіть знак операції");
  znak=getche();
  switch(znak)
   { case '+': y = a+b; break;
     case '–': y = a–b; break;
     case '*': y = a*b; break;
```



```
     case '/': y = a/b; break;
     case '%': y = a%b; break;
     default: printf ("\nНевірна операція");
     break;
   }
  if ((znak = = '+')||(znak = ='–')||(znak = ='*')||(znak = ='/')||(znak = ='%'))
     printf ("a %c b = %d", znak,y);
}
```

Синтаксично конструкції *case* і *default* є мітками. Вони істотні лише при початковій перевірці, коли вибирається оператор для виконання в тілі *switch*. Всі оператори тіла, наступні за вибраним, виконуються послідовно, ніби не помічаючи міток *case* і *default*, якщо тільки який-небудь з операторів не передасть керування за межі тіла оператора *switch*. Для виходу з тіла оператора звичайно використовується оператор *break*.

Оскільки ключове слово *case* разом з константою слугує просто міткою і якщо будуть виконуватись оператори для якогось варіанта *case*, то далі виконуватимуться оператори всіх наступних варіантів, поки не зустрінеться *break*, що здійснює вихід з оператора *switch* до наступного за ним оператора.

В операторі *switch* не можуть повторюватися константи, але з різними константами можуть пов'язуватися одні й ті самі оператори. У цьому разі допустима запис:

```
switch (<вираз>)
{ case <константа 1>:
  case <константа 2>:
  { <оператор 1>;
...
    <оператор n>;
    [break; ]}
  [default: ]
  {[<оператор n>;]
  [break; ]}
}
```

У разі вибору варіантів на основі обчислення та аналізу змінної з плаваючою крапкою оператор *switch* використовувати не можна, не можна використовувати даний оператор і тоді, коли можливі значення змінних, що аналізуються, потрапляють у певний діапазон, тобто оператор *switch* може виконувати тільки операції строгої рівності.

Не може бути двох констант в одному операторі *switch*, що мають однакові значення. Але оператор *switch*, що включає в себе інший оператор *switch*, може містити аналогічні константи.

Вкладені оператори *switch*. Оператор *switch* може мати серед послідовності операторів інший оператор *switch*. Якщо константи *case* внутрішнього і зовнішнього оператора мають однакові значення, не виникає конфліктної ситуації.

Наприклад,

```
switch (ch)
{ case 1:
    switch (x)
     { case 0: printf ("Ділення на 0, помилка");
               break;
       case 1: error(fl);
               break;
     }
 case 2:
...
}
```

7.6. ОПЕРАТОР ПЕРЕХОДУ

Оператор переходу використовується для зміни природної послідовності виконання операторів програми. Формат оператора є таким:

```
goto <мітка>;
...
<мітка>: <оператор>;
```

де <мітка> — ідентифікатор, який записується перед оператором програми, якому необхідно передати керування. Мітка відділяється від оператора двокрапкою. Наприклад,

```
k = k + 1;
if (k > 5)
   goto A;
b = b*k;
A: b = b/k;
```

Прийнята в цей час дисципліна програмування рекомендує або зовсім не використовувати *goto*, або звести його застосування до мінімуму і дотримуватися таких правил:

- не входити всередину блока поза блоком;
- не входити всередину умовного оператора *if*, тобто не передавати керування операторам, розміщеним після ключових слів *if* або *else*;

- не входити ззовні всередину оператора *switch*;
- не передавати керування всередину оператора циклу;
- не входити в блок ззовні.

Рекомендується використати даний оператор лише в тому разі, коли необхідно передати керування в програмі «згори вниз», забороняється використовувати оператор для передачі керування «знизу вгору»

7.7. СКЛАДЕНИЙ І ПОРОЖНІЙ ОПЕРАТОРИ

Порожній оператор — оператор, який складається тільки з символу крапка з комою ; . Він виконується там, де за правилами синтаксису потрібен оператор, але за логікою програми оператор повинен бути відсутнім. Необхідність використання порожнього оператора виникає у таких випадках:

1) в операторах циклу *do, for, while* та умовному операторі *if*, коли тіло оператора не потрібно, хоча за синтаксисом потрібен хоча б один оператор. Наприклад, необхідність у використанні порожнього оператора виникає при програмуванні циклів, коли дії, які можуть бути виконані в тілі циклу, цілком вміщуються в заголовок циклу. Обчислити факторіал 5!.

```
for (i = 0, p = 1; i < 5; i + 1, p* = i)
 ;
```

Тіло циклу складається з порожнього оператора, оскільки немає потреби в інших операторах.

2) при необхідності позначити фігурну дужку міткою.

Порожній оператор, подібно до будь-якого іншого оператора мови С++ може бути позначений міткою. Наприклад, щоб позначити фігурну дужку складеного оператора, яка не є оператором, потрібно вставити перед нею позначений порожній оператор.

```
main ( )
{...
  for (...){
   for (...){
    while (...){
     if (...) goto end;
              }
        }
     }
end:;
}
```

Складений оператор. Будь-яка послідовність операторів, узятих у фігурні дужки, є *складеним оператором*. Він не повинен закінчуватися крапкою з комою, бо ж обмежувачем блока слугує сама дужка. Всередині блока кожен оператор повинен закінчуватися крапкою з комою.

Формат оператора:

```
{ [оголошення];
  оператор;
  [оператор n];
 }
```

Усі оголошення, вміщені у складений оператор, повинні бути на початку. Виконання складеного оператора полягає в послідовному виконанні операторів, з яких він складається. Типове використання складеного оператора як тіла іншого оператора, наприклад, оператора *if*.

```
if (i> = 0)
{ int j = i + 1;
  n+ = j /m;
}
```

Змінні, які описані всередині складених операторів, можуть бути використані лише всередині блока. Змінна j всередині дужок не визначена.

Оператори, що входять у складовий оператор, у свою чергу можуть бути складеними.

7.8. ДИРЕКТИВИ ПРЕПРОЦЕСОРА, ЇХ ВИКОРИСТАННЯ У ПРОГРАМАХ

Отримання програми, що виконується з початкового тексту мовою С++, відбувається в кілька етапів. На найпершому етапі з початковим текстом програми працює спеціальна програма — препроцесор. Препроцесор — програма, що використовується для обробки початкового файла на нульовій фазі компіляції і замінює символічні абревіатури в програмі на відповідні директиви. Він відшукує інші файли, необхідні для виконання програми, і може змінити умови компіляції. Основна мета препроцесора — закінчити остаточний текст програми на С++. Потім остаточний текст програми зазнає компіляції. Директиви препроцесора — інструкції, записані в початковому

тексті програми мовою С++ і призначені для виконання препроцесором мови С++. Препроцесор мови аналізує програму до компілятора і заміщує символічні абревіатури у програмі на відповідні директиви. Директиви препроцесора називають ще командними рядками, вони починаються зі спеціального знака #, що вміщується в першій позиції рядка. Директиви препроцесора дозволяють:

1) описувати макро, які зменшують трудомісткість написання програми і поліпшують зрозумілість початкового тексту програми. Використання макро замість функцій збільшує швидкість виконання програми за рахунок виключення втрат часу на виклик функцій, але за рахунок збільшення розміру завантажувального модуля програми;

2) включати текст з інших текстових файлів, що містять прототипи бібліотечних і написаних користувачем функцій, шаблони структурних змінних, об'єднань, перелічувальних типів, зовнішніх змінних, символічних констант. Це підвищує мобільність програм при переході з однієї системи програмування на іншу;

3) організувати умовну компіляцію, тобто в залежності від заданих у командному рядку або середі параметрів отримувати різний програмний код, що поліпшує переносимість і налагодження коду.

Найчастіше використовуються директиви препроцесора *#define* та *#include*.

#define — може бути записана в будь-якому місці програми і має силу від місця появи до кінця файла. Директива використовується для визначення символічних констант. Формат директиви є таким:

#define <макровизначення> <рядок заміщення>

Як макровизначення можуть використовуватися константи, змінні та вирази. Макровизначення може бути з аргументами, які беруться у круглі дужки. За синтаксисом макровизначення з аргументами збігається з функцією, але за механізмом дії в них є суттєва відмінність. При виклику функції значення аргументу передається під час виконання програми, при виклику макровизначення рядок аргументів передається у програму до компіляції, таким чином змінити значення аргументу неможливо. Не дозволяється у макровизначенні використовувати пробіли, бо препроцесор сприймає їх як початок рядка заміщення.

Схема дій директив така: щойно в тексті програми зустрінеться макровизначення, воно тут же замінюється «рядком заміщення». Наприклад,

```
#define k 10
#define pr "значення результату = %d\n"
...
if ( i<k) m = m + 1;
...
printf (pr, m);
...
```

Для скасування директиви #define існує директива #undefine. Вона має ось який вигляд:

```
#undef <макровизначення>
#define KL 100
#undef KL // KL тепер не визначений
```

У мові C++ існують стандартні макровизначення, які не потрібно визначати в директиві #define, але можна використовувати за замовчуванням, наприклад

__FILE__ __TIME__ __LINE__ __DATE__ __STDC__

Вони не можуть бути піддані впливу директиви #undef, їх не можна перевизначити.

Директива #include відшукує наступне за ключовим словом #include ім'я файла і включає його в текст початкової програми. Формат директиви є таким:

```
#include <ім'я файла>
#include "ім'я файла"
```

Якщо ім'я файла задане в кутових дужках, то пошук файла, що включається, здійснюється в стандартних каталогах системи. Якщо ім'я файла задане в лапках, то пошук спочатку проводиться в робочому каталозі, а потім уже в стандартних.

Директива #include не обмежується лише включенням заголовного файла. За допомогою #include можна підключати С-модулі, які необхідно компілювати сумісно з початковою програмою. Якщо, наприклад, окремий модуль вашої програми має ім'я sort.cpp, то можна, використовуючи директиву #include "sort.cpp", компілювати її спільно з основною програмою.

Окрім названих директив, також використовуються: #undefine (відмінити визначення).

Директиви #if, #else, #endif відповідають використанню логічного оператора *if*.

```
#if вираз 1
 // послідовність операторів
#elif вираз 2
 // послідовність операторів
#elif вираз 3
```



```
 // послідовність операторів
...
#else
 // послідовність операторів
#endif
```

Директива #elif означає "інакше якщо" і використовується для побудови конструкції if-else-if. За elif слідує константний вираз. Приклад використання директив умовної компіляції.

```
#define US 0
#define ENGLAND 1
#define FRANCE 2
#define COUNTRY US
#if COUNTRY== US
   char currency[] = "dollar";
#elif COUNTRY ==ENGLAND
   char currency[] = "pound";
#else
   char currency[] = "franc";
#endif
```

7.9. ЗУМОВЛЕНІ МАКРОСИ

Borland C++ зумовлює такі глобальні ідентифікатори. За винятком __cplusplus і _WINDOWS, кожен з них починається і закінчується подвійним підкресленням ("__").

__BCOPT__ — визначений в 1, якщо використовується оптимізація.

__BCPLUSPLUS__ — визначений, якщо вибрана компіляція програми як C++ -програм.

__BORLANDC__ — містить поточну версію компілятора.

__CDECL__ — встановлений в 1, якщо використовується стандартна угода про виклик функцій у C і C++; інакше не визначений.

__CHAR_UNSIGNED — визначений, якщо стандартним символьним типом стає unsigned.

__CONSOLE__ — визначений, якщо програма є консольним додатком.

__cplusplus — встановлений в 1 для 32-розрядного компілятора, якщо програма буде компілюватися як C++ -програма, в інших випадках не визначений.

__DATE__ — містить рядок символів у вигляді місяць/день/рік, відповідний даті компіляції програми.

__DLL__ — встановлений в 1, якщо компілюється програма для Windows DLL; інакше не визначений.

__FILE__ — містить ім'я початкового файла, що компілюється у вигляді символьного рядка. Цей макрос змінюється, коли компілятор обробляє директиву #include або #line або коли включення файла завершено.

__LINE__ — видає номер поточного рядка початкового файла, що обробляється C++, у вигляді десяткової константи. Звичайно перший рядок початкового файла визначений як 1, але директива #line може змінити це.

_M_IX86__ — визначений завжди.

__MSDOS__ — встановлений у цілу константу 1 для компіляцій MS-DOS.

__MT__ — встановлений в 1, якщо багатопоточна бібліотека використовується з 32-розрядним компілятором.

__OVERLAY__ — встановлений в 1, якщо компілюється модуль з опцією –Y (підтримка оверлей включена); інакше не визначений.

__PASCAL__ — встановлений в 1, якщо при компіляції програми використовується угода про виклики функції Паскалю; інакше не визначений.

__STDC__ — встановлений в 1, якщо компіляція програми відбувається з використанням стандарту ANSI C з перевіркою на сумісність, в іншому разі не визначений.

__TCPLUSPLUS__ — шістнадцяткове значення, яке відповідає версії компілятора.

__TEMPLATES__ — встановлений в 1 для всіх версій компіляторів, що підтримують шаблони.

__TIME__ — містить час початку компіляції програми в форматі години:хвилини:секунди.

__TLS__ — визначений не нульовим значенням для 32-розрядного компілятора.

__TURBOC__ — визначений як шістнадцяткова константа, збільшується з кожною новою версією.

_WCHAR_T — встановлений в 1 для зазначення того, що тип wchar_t є вбудованим типом даних.

_WCHAR_T_DEFINED — встановлений в 1 для зазначення того, що тип wchar_t є вбудованим типом даних.

_Windows — визначений, якщо компілюється програма для Windows.

__WIN32__ — встановлений в 1 для 32-розрядного компілятора.

Розділ 8

ФОРМАТОВАНЕ ВВЕДЕННЯ-ВИВЕДЕННЯ C++

8.1. ОПЕРАЦІЯ *ПОМІСТИТИ В ПОТІК* І ОПЕРАЦІЯ *ВЗЯТИ З ПОТОКУ*

C++ підтримує всі функції введення-виведення C і визначає свою власну об'єктно-орієнтовану систему введення-виведення. Щоб забезпечити доступ програмі до бібліотеки потоків, необхідно підключити заголовний файл ***<iostream.h>***, у якому визначені складні набори ієрархії класів, що підтримують операції введення-виведення. Клас ***ios*** забезпечує підтримку форматованого введення-виведення, контроль помилок і інформацію про стан потоку введення-виведення. Від нього породжені класи ***istream, ostream, iostream***, які використовуються відповідно для створення потоків введення, виведення і введення-виведення.

Операція лівого зсуву використовується в C++ як операція *помістити в потік*, а операція правого зсуву використовується як операція *взяти з потоку*. Кожна з цих операцій перевантажена в бібліотеці класів C++. Таке перевантаження дозволяє використати одноманітний синтаксис для введення і виведення, символів рядків, цілого і дійсних чисел. У загальному випадку для виведення на екран монітора використовується така форма операції <<:

cout << *вираз*;

де *вираз* може бути будь-яким виразом C++, включаючи інші вирази виведення;

cout — стандартний потік виведення, тобто звичайно на екран, але може бути зв'язаний і з іншим пристроєм.

У загальному випадку для введення значень з клавіатури використовується така форма операції >>:

cin >> *змінна;*

де cin — стандартний потік введення, звичайно з клавіатури, але може бути зв'язаний і з іншим пристроєм.

Кожна з перевантажених операцій << і >> може бути використана в зчепленій формі, тобто в одному виразі введення-виведення можна виводити більше за одну величину. Наприклад,

```
// обчислення суми двох цілих чисел
#include <iostream.h>
int main()
```

```
{ int x, y;
  cout << "Введіть два цілих числа: ";
  cin >> x >> y;
  cout << "Сума чисел" << x << "і" << y "рівна" << (x + y) <<
endl;
  return 0;
}
```

Як видно з цього прикладу, можна ввести в одному операторі кілька елементів даних. Під час виведення результату на екран потрібно включати пробіли між елементами даних для зручності читання інформації, що виводиться.

8.2. ФОРМАТОВАНЕ ВВЕДЕННЯ-ВИВЕДЕННЯ

Бібліотека потоків C++ передбачає форматування виведення: використання функцій-членів класу *ios*, прапорів і маніпуляторів.

8.2.1. Прапори форматування

Кожен потік введення-виведення пов'язаний з набором прапорів формату, які керують способом форматування інформації і являють собою бітові маски. Ці маски оголошені в класі *ios* як дані перелічувального типу, ці значення необхідні для встановлення або скидання прапорів формату. Прапори форматування введення-виведення перераховані в табл. 8.1.

Коли прапор *skipws* скинений, невидимі символи не відкидаються. Якщо прапори *left, right, internal*, то за замовчуванням використовується вирівнювання по правому краю. За замовчуванням числові значення виводяться в десятковій системі числення. Якщо жоден з прапорів *scientific* або *fixed* не встановлений, то компілятор сам вибирає відповідний спосіб виведення. За замовчуванням коли встановлений прапор *fixed*, для десяткових цифр виводиться шість позицій.

Керують установками прапорів функції *setf, unsetf* і *flags*.

Для встановлення прапорів використовується функція *setf( )*, яка має такий формат:

long setf (long flags);

Таблиця 8.1

ПРАПОРИ ФОРМАТУВАННЯ ВВЕДЕННЯ-ВИВЕДЕННЯ

Прапор	Призначення
skipws	Пропускає при введенні пробіли, символи табуляції і нового рядка.
left	Вирівнювання виведення по лівому краю.
right	Вирівнювання виведення по правому краю.
internal	Для заповнення поля виведення відбувається вставка пробілів між усіма цифрами і знаками числа.
dec	Виведення здійснюється в десятковій системі числення.
oct	Виведення здійснюється у вісімковій системі числення.
hex	Виведення здійснюється в шістнадцятковій системі числення.
showbase	Показувати при виведенні основу системи числення.
showpoint	Для дійсних чисел виводить десяткову крапку і подальші нулі.
uppercase	При виведенні використовує букви верхнього регістра: символ X і букви A, B, C, D, E, F для шістнадцяткових чисел; символ порядку E для вісімкових чисел.
showpos	Додатні числа виводяться зі знаком +.
scientific	Використати експоненціальну (наукову) нотацію для дійсних чисел.
fixed	Для дійсних використати представлення в форматі з фіксованою крапкою.
unitbuf	Буфер очищається після кожної операції вставки.
stdio	Очищає потоки stdio, sterr після кожного виведення.
boolalpha	Значення булевого типу виводяться у вигляді ключових слів true і false

При модифікації основи системи числення можна використати прапор *basefield* як другий параметр функції *setf( )*, це дозволяє встановити один з прапорів *oct, dec* і *hex*, якщо жоден з цих прапорів не встановлений, то за замовчуванням числа трактуються як десяткові.

При завданні способу вирівнювання можна використати прапор *adjustfield* як другий параметр функції *setf( )*, це гарантує, що функція *setf( )* встановить лише один з трьох прапорів *left, right, internal.*

Прапори *adjustfield, basefield, floatfield* зручно використати в тому разі, коли перед установкою прапора потрібно скинути всі прапори, які не можуть бути одночасно з ним встановлені.

При завданні нотації дійсних чисел можна використати прапор *floatfield* як другий параметр функції *setf( )*, виклик cout.setf (0, ios::floatfield) відновлює в системі формат виведення з плаваючою крапкою за замовчуванням.

Дана функція повертає попереднє значення прапора, встановлюючи його нове значення рівним значенню, заданому параметром *flags*. При цьому всі інші прапори залишаються незмінними. Наприклад, для того, щоб встановити прапор *showpos*, можна використати таку інструкцію:

потік_введення-виведення. setf (ios::showpos);

де *потік_введення/виведення* — це потік, на який необхідно впливати.

cout. setf (ios::showpos);

Оскільки прапори визначені в класі *ios*, вони використовуються із зазначенням імені класу та операцією дозволу області видимості (::), тобто сам прапор *showpos* задати не можна, необхідно написати *ios::showpos*. Слід пам'ятати, що функція *setf( )* є членом класу *ios* і впливає на створені цим класом потоки введення-виведення, тому будь-який виклик функції *setf( )* робиться відносно конкретного потоку. Не можна викликати функцію *setf( )* саму по собі. Кожний потік введення-виведення підтримує власну інформацію про стан формату.

Замість повторних викликів функції *setf( )* в одному виклику можна встановити відразу кілька прапорів, для об'єднання необхідних прапорів використовується операція порозрядного АБО. Наприклад,

cout.setf (ios::hex ¦ ios::scientific);

Для відключення прапорів потрібно використати функцію *unsetf( )*. Прототип даної функції має вигляд:

long unsetf (long flags);

Функція повертає значення попередньої установлення прапорів і скидає прапори, визначені параметром *flags*. Наприклад,

cout.unsetf (ios::scientific);

Для того, щоб узнати поточні установки прапорів і при цьому нічого не міняти, використовується функція *flags( )*, яка має такий прототип:

long flags ( );

Ця функція повертає поточний стан прапорів потоку. Функція *flags( )* має другу форму, яка дозволяє встановити прапорам значення, що повідомляється параметром і повернути попереднє значення прапорів. Прототип функції такий:

long flags (long flags);

8.2.2. Функції width( ), precision( ), fill( )

Крім прапорів у класі *ios* визначені три функції-члени, які дозволяють встановити ширину поля потоку, символ для заповнення і число цифр після десяткової крапки.

int width (int len);

Функція *width( )* повертає поточну ширину поля потоку і встановлює нове мінімальне значення, що визначається параметром *len*. Якщо значення, що виводиться, має меншу ширину, ніж задана ширина поля, то зайві позиції заповнюються поточним символом заповнення, за замовчуванням визначений пробіл. Якщо значення, що виводиться, перевершує мінімальну ширину поля, то воно усікається, значення *len* ігнорується і значення виводиться повністю. Якщо функція використовується під час введення, то параметр задає максимальне число символів, що читаються.

int width ( );

Функція *width( )* без параметрів повертає поточне значення змінної ширини поля потоку.

Для читання або зміни поточного символу заповнення можна використати функцію *fill*.

char fill ( );

Дана функція повертає поточний символ заповнення.

char fill (char ch);

Ця функція повертає колишнє значення символу заповнення і символ *ch* стає новим символом заповнення. Символ заповнення використовується для заповнення порожніх місць відповідно до заданої специфікації ширини поля.

За замовчуванням при виведенні з плаваючою крапкою точність дорівнює шести цифрам, однак це число може бути змінено за допомогою функції *precision*. Прототип функції є такими:

int precision (int num);

int precision ( );

Перша функція встановлює точність (число цифр, що виводяться після крапки), задану параметром num, і повертає колишнє значення точності. Нижче наведена програма, що демонструє використання цих функцій.

```
#include <iostream.h>
int main ( )
{ cout.setf (ios::scientific); //число виводиться в науковій нотації
  cout << 987.65 << endl;
  cout.setf (ios::fixed);      //число виводиться в звичайній нотації
  cout.width (10);             //встановлення ширини поля
```

```
  cout.precision (3);       //встановлення точності в 3 цифри
  cout << 987.654321 <<endl;
  cout.width (9);
  cout.fill ('*');          //встановлення символу заповнення *
  cout.precision (2);       //встановлення точності в 2 цифри
  cout << 987.654321;
  return 0;
}
```

Після виконання програми на екран буде виведено:

```
9.876500e+02
    987.654
***987.65
```

8.2.3. Маніпулятори введення-виведення

Система введення-виведення C++ надає ще один спосіб зміни параметрів форматування потоку. Цей спосіб використовує спеціальні функції, які називаються маніпуляторами. Маніпулятори введення є в деяких випадках зручнішими, ніж прапори та функції формату класу ios. Маніпулятори можуть бути включені у вирази введення-виведення. Наприклад,

```
cout<<setw(6)<<x<<setw(4)<<y;
```

еквівалентно

```
cout.width(6);
cout << x;
cout.width(4);
cout << y;
```

Розрізняють прості і параметризовані маніпулятори, які вимагають вказівки параметра. Для доступу до маніпуляторів з параметрами необхідно в програму включити заголовний файл ***<iomanip.h>***. Маніпулятори наведені в табл. 8.2.

Маніпулятор впливає лише на потік, частиною якого є вираз введення-виведення, що містить маніпулятор, вони не впливають на всі відкриті в даний момент потоки. Маніпулятор *setbase* встановлює основу системи числення. Значення параметра *base* можуть бути: 0, 8, 10 і 16. При використанні параметра 0, основа за замовчуванням, при виведенні використовується десяткова основа, при введенні числа, що починається з '0', вважається вісімковою, числа, що починається з '0*x*' — шістнадцятковими. При використанні параметра 8 вважається, що використовується вісімкова основа, при 10 — десяткова і при 16 — шістнадцяткова. Інші основи ігноруються.

Таблиця 8.2

ПРОСТІ І ПАРАМЕТРИЗОВАНІ МАНІПУЛЯТОРИ

Маніпулятор	Опис
boolalpha	При введенні і виведенні встановлення прапора boolalpha.
dec	При введенні і виведенні встановлює прапор десяткової системи числення.
endl	Виведення нового рядка і чищення потоку.
ends	Виведення нульової ознаки кінця рядка.
fixed	При виведенні встановлення прапора fixed.
flush	Очищає вихідний потік (скидає його буфер).
hex	При введенні і виведенні встановлює прапор шістнадцяткової системи числення.
internal	При виведенні встановлення прапора internal.
left	При виведенні встановлення прапора left.
noboolalpha	При введенні і виведенні скидання прапора boolalpha.
noshowbase	При виведенні скидання прапора showbase.
noshowpoint	При виведенні скидання прапора showpoint.
noshowpos	При виведенні скидання прапора showpos.
noskipws	При введенні скидання прапора skipws.
nounitbuf	При виведенні скидання прапора unitbuf.
nouppercase	При виведенні скидання прапора uppercase.
oct	При введенні і виведенні встановлює прапор вісімкової системи числення.
resetiosflags (long f)	При введенні і виведенні скидає прапори, задані параметром f.
right	При виведенні встановлення прапора right.
scientific	При виведенні встановлення прапора scienfitic.
setbase (int base)	Встановлює основу системи числення.
setfill (int ch)	При виведенні встановлює символ заповнення, що дорівнює ch.
setiosflags (long f)	При введенні і виведенні установка прапора, заданого параметром f.

Закінчення табл. 8.2

Маніпулятор	Опис
setprecision (int p)	При виведенні задає точність для дійсних чисел.
setw (int w)	Встановлює ширину поля, яка дорівнює w.
showbase	При виведенні встановлення прапора showbase.
showpoint	При виведенні встановлення прапора showpoint.
showpos	При виведенні встановлення прапора showpos.
skipws	При введенні встановлення прапора skipws.
unitbuf	При виведенні встановлення прапора unitbuf.
uppercase	При виведенні встановлення прапора uppercase.
ws	При введенні ігнорує початкові символи пробілу.

Наприклад,

```
#include <iostream.h>
#include <iomanip.h>
void main()
{ cout << setiosflags (ios::fixed);
  cout << setprecision(2) << 500.6754 << endl;
  cout << setfill ('#') << setprecision(4) << 1.23456;
}
```

Розділ 9

МАСИВИ ДАНИХ

9.1. ОГОЛОШЕННЯ ТА ІНІЦІАЛІЗАЦІЯ МАСИВУ

Масив є структурованим типом даних. Масив — це сукупність елементів одного типу, які використовуються в програмах під одним ім'ям. Кожен масив має ім'я, яке повинно відповідати тим же правилам, що й імена змінних.

Доступ до окремих елементів масиву здійснюється за іменем масиву та індексом (порядковим номером) елемента, який вказує відносну позицію елемента. Індекс — це число, за допомогою якого розрізняються елементи масиву. Елементи — це окремі змінні в масиві. Класичними прикладами масивів є вектор і матриця. Основні властивості масиву:

1) всі елементи масиву мають один і той самий тип;

2) усі елементи масиву розташовані в пам'яті один за одним. Індекс першого елемента дорівнює нулю;

3) ім'я масиву є вказівником-константою, що дорівнює адресі початку масиву (першого байта першого елемента масиву).

Для роботи з масивом у програмі необхідно за аналогією з простими змінними зробити його оголошення на початку головної функції чи блоку. Формат оголошення масиву є таким:

<тип даних> <ім'я масиву> [розмірність масиву];

Тип даних при описі масивів задає тип елементів масиву і вибирається за тими ж правилами, що й для простих змінних.

Ім'я масиву формується за тими самими правилами, що й імена простих змінних.

Розмірність масиву — це число індексів, що використовуються для посилання на конкретний елемент масиву. Розмірність масиву описується для кожного індексу окремо. Вона визначає кількість значень кожного індексу і повинна бути задана константою, вказаною явно в квадратних дужках, або повинна бути визначена за допомогою директиви #define для автоматичних змінних. Нумерація елементів масиву починається з нуля, тому, якщо в одновимірному масиві 50 елементів, то верхня межа зміни індексу дорівнює 49. Тобто перший елемент масиву array означається як array[0], другий array[1], останній array[49]. Якщо двовимірний масив має 10 рядків і 10 стовпців, то індекс рядка

прийматиме верхнє значення індексу, себто 9, і стовпця — 9. Рис. 9.1 ілюструє зміну індексу за рядками і стовпцями в двовимірному масиві.

	Стовпець 0	Стовпець 1	Стовпець 2	Стовпець 3
Рядок 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Рядок 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Рядок 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Рис. 9.1. Двовимірний масив з трьома рядками і чотирма стовпцями

Приклад оголошення масивів:

```
#define k 10
int main()
{ float a[k]; // оголошення масиву a з 10 елементів типу float
  int b[10][10]; // оголошення двовимірного масиву зі 100 еле-
                 // ментів цілого типу
  int c [9]; // оголошення одновимірного масиву з 9 елементів
             // цілого типу
....
}
```

C++ підтримує багатовимірні масиви. Найпростішим видом багатовимірного масиву є двовимірний масив, який можна представити як масив одновимірних масивів. Двовимірний масив являє собою матрицю, де перший індекс відповідає за рядок, а другий за стовпець. Кожна розмірність масиву вміщується в окремі квадратні дужки. Багатовимірний масив оголошується таким чином:

<тип даних> <ім'я масиву> [розмірність N]...[розмірність 2] [розмірність 1];

Елементи багатовимірного масиву зберігаються в пам'яті в порядку зростання найправішого індексу, тобто за рядками. Це означає, що правий індекс змінюється швидше лівого, якщо переміщатися масивом у порядку розташування елементів в пам'яті.

Перед тим як використати масив, необхідно присвоїти значення його елементам. У C++ масиви не ініціалізуються і не обнулюються автоматично. Елементам масиву можна задати початкові значення трьома способами: ініціалізацією, присвоєнням або введенням. При описі масиву може бути виконана ініціалізація елементів масиву.

Є два методи ініціалізації:

1) *ініціалізація за замовчуванням.* Якщо не проводиться ініціалізація елементів масиву, то всі елементи статичних та зовнішніх масивів ініціалізуються компілятором нулями, а елементи автоматичних і регістрових масивів визначені на неочищену пам'ять;

2) *явна ініціалізація елементів.* Після опису масиву можна записати список початкових значень масиву, які беруться у фігурні дужки.

Існують дві форми явної ініціалізації:

1) явне зазначення числа елементів масиву та список початкових значень, можливо, з меншим числом елементів.

<тип даних> <ім'я масиву> [розмірність N]...[розмірність 1] = {<список значень>};

Наприклад, ініціалізація зовнішнього масиву з 10 елементів:

int array [10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

Якщо оголошений масив b з 10 елементів

int b [10] = {1, 2, 3, 4};

то перші чотири елементи масиву ініціалізувати числами 1, 2, 3 і 4. Значення інших шести елементів або дорівнює 0, якщо масив зовнішній чи статистичний, або не визначено, якщо масив автоматичний чи регістровий. При використанні автоматичних масивів програміст повинен присвоїти нульове початкове значення принаймні першому елементові для того, щоб автоматично були обнулені елементи, що залишилися. При ініціалізації багатовимірних масивів можна додати фігурні дужки навколо кожного вимірювання. Наприклад,

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

Якщо даних рівно стільки, скільки має бути при заповненні всіх елементів, то фігурні дужки в списку можуть бути опущені, тоді наступні два записи еквівалентні:

int f [2][2] = {{1, 2}, {3, 4}}; int f [2][2] = {1, 2, 3, 4};

Завдання в списку початкових значень більшого числа значень, ніж є елементів у масиві, є синтаксичною помилкою;

2) без явного зазначення елементів масиву, тільки зі списком початкових значень. Компілятор визначає число елементів масиву за списком ініціалізації.

<тип даних> <ім'я масиву> [ ][розмірність N–1]...[розмірність 1] = {<список значень >};

Наприклад,

int array [ ] = {1, 2, 3, 4};

У результаті створюється масив з чотирьох елементів, і ці елементи отримують початкові значення зі списку ініціалізації.

Масиву символів можна задавати початкові значення, використовуючи рядкову константу. Наприклад, оголошення

```
char c[ ] = {"ABCD"};
```

присвоює елементам масиву *c* початкові значення як окремі символи рядка "ABCD". Розмір масиву *c* визначається на основі довжини рядка плюс нульовий символ закінчення рядка, таким чином масив містить п'ять елементів. Символьний масив можна задати окремими символьними константами. Попереднє оголошення еквівалентне наступному:

```
char c [ ] = {'A', 'B', 'C', 'D', '\0'};
```

Для багатовимірних масивів необхідно визначити всю розмірність, крім найлівішої, компілятор визначає число елементів за числом значень у списку ініціалізації. Наприклад,

```
int array [ ] [3] = {0, 1, 2, 3, 4, 5};
```

тобто кожен рядок міститиме три елементи, нульовий — 0, 1, 2; перший — 3, 4, 5. Якщо потрібно проініціалізувати не всі елементи, в списку ініціалізації використовуються фігурні дужки, наприклад:

```
int array [ ][3] = {{0}, {3, 4}};
```

m [0][0] дорівнюватиме 0, m [1][0] — 3, m [1][1] — 4.

Використовуючи операцію присвоєння, можна в програмі присвоїти значення кожному елементу масиву або, використовуючи оператор циклу і функції введення, можна вводити значення елементів з клавіатури. Ці способи ініціалізації масиву будуть показані нижче.

9.2. ДОСТУП ДО ЕЛЕМЕНТІВ МАСИВУ

Доступ до елементів масиву може виконуватися за допомогою операції індексації або за допомогою механізму вказівників. У першому випадку для звернення до елементів масиву достатньо в потрібному місці вказати ім'я масиву та поточний номер індексу. При використанні операції індексації для посилання на необхідний елемент вказується його номер у масиві, вміщений у [ ].

У мові C++ дозволяється лише поелементне звертання до масиву, при якому поточне значення може бути задане константою, змінною чи виразом. Наприклад,

```
int a[9]; a[0] = 0; a[1] = 0; a[2] = 0; a[3] = 0; a[4] = 0; a[5] = 0;
a[6] = 0; a[7] = 0; a[8] = 0;
```

У даному прикладі обнуляються елементи масиву a. Операції з багатовимірними масивами зручно виконувати, використовуючи оператор циклу *for*. Наступний фрагмент програми показує, як обнулити елементи двовимірної матриці.

```
int b[9][9], i, j;
for (i = 0; i < 9; i++)
  for (j = 0; j < 9; j++)
    a[i][j] = 0;
```

Інший спосіб доступу до елементів масиву — використання механізму вказівників. Ім'я масиву, що використовується без наступних [ ], являє собою адресу початку масиву. Оскільки ім'я масиву — це вказівник-константа на перший байт першого елемента масиву, то, використовуючи операцію отримання значення за адресою (*), можна виконати доступ до будь-якого елемента масиву. Тобто будуть еквівалентними запис до посилання на i-й елемент масиву array array[i] і *(array+i).

Нехай є опис масиву int a[5], який містить 5 елементів: a[0], a[1], a[2], a[3], a[4]. Адреса i-го елемента масиву дорівнює сумі адреси початкового елемента масиву і зсування цього елемента на і одиниць від початку масиву. Якщо ра — це вказівник на ціле int *ра; , то після виконання оператора ра = &a[0]; ра містить адресу елемента a[0]. Вираз ра+1 вказує на наступний елемент, ра+i вказує на i-й елемент масиву, тобто є адресою a[i], тоді *(ра+i) є вмістом i-го елемента. Оскільки ім'я масиву ототожнюється з адресою його першого елемента, то оператор ра = &a[0]; еквівалентний оператору ра = a; , тому будуть еквівалентними запис до посилання на i-й елемент масиву a[i] і *(a+i). Будь-який масив і індексний вираз можна представити за допомогою вказівника. В той же час між ім'ям масиву й відповідним вказівником є істотна відмінність. Треба пам'ятати, що вказівник — це змінна, а ім'я масиву — константа. Тому ра = a; і ра++; допустимі операції. Оператори вигляду a = ра; a++; використати не можна, оскільки значення константи постійне і не може бути змінене.

Якщо до вказівника додається ціле, компілятор автоматично масштабує ціле, множачи його на число байтів, відповідне типу, зазначеному в оголошенні вказівника. Розміщення масиву a в пам'яті подано на рис. 9.2, якщо він розташовується з адреси 1000.

1000	1002	1004	1006	1008
*ра	*(ра+1)	*(ра+2)	*(ра+3)	*(ра+4)
a[0]	a[1]	a[2]	a[3]	a[4]
*a	*(a+1)	*(a+2)	*(a+3)	*(a+4)

Рис. 9.2. Розміщення масиву в пам'яті

Тоді вираз pa + 3 або a + 3 дає значення адреси 1000 + 3*sizeof(int) = 1000 + 3*2 = 1006. Взявши значення за цією адресою, можна набути значення четвертого елемента масиву, тобто *(pa+3) або *(a+3). Обидва способи еквівалентні.

При роботі з двовимірними масивами можна також використати вказівники. У цьому разі точкою відліку може бути як найперший елемент масиву, так і перший елемент кожного з рядків. Передбачимо, що є оголошення

```
int array[4][2];     // масив array типу int з 4 рядків і 2 стовпців
int *pa;             // вказівник pa на цілий тип
```

pa=&array[0][0]; // вказівнику pa присвоїти адресу першого елемента масиву a[0][0]

pa = &array[0][0]; можна записати pa = array; тобто array = = &a[0][0];.

Двовимірний масив розташовується в пам'яті подібно до одновимірного масиву, за рядками, спочатку перший рядок, потім другий, третій і т. д., послідовно займаючи елементи пам'яті. Таким чином,

pa = = &array[0][0];
pa + 1 = = &array[0][1];
pa + 2 = = &array[1][0];
pa + 3 = = &array[1][1];

Тоді pa + 6 вказуватиме на елемент array[3][0].

Якщо використовується вказівник на перший елемент рядка, то масив представляється як масив масивів. Якщо масив array з попереднього прикладу має чотири рядки, кожен з яких є масивом з двох елементів. Ім'я першого рядка array[0], ім'я другого array[1] і т. д. Проте ім'я масиву є також вказівником на цей масив, тому,

array[0] = = &array[0][0];
array[1] = = &array[1][0];
array[2] = = &array[2][0];
array[3] = = &array[3][0];

Тоді доступ до елемента масиву, якщо використовується вказівник на перший елемент рядка, можна здійснити за допомогою виразу (array[i]+j), а щоб набути значення, розташованого за цією адресою, необхідно використати вираз *(array[i]+j). Якщо ім'я масиву використовується як адреса початку масиву вказівників, то вирази *(*(array+i)+j) і array[i][j] є еквівалентними.

Багатовимірний масив у мові С++ — це масив масивів, тобто елементами якого є масиви. Двовимірний масив можна розглядати як одновимірний масив, елементами якого є рядки масиву. Нехай дана матриця a, яка має m рядків і n стовпців. Упорядкування елементів матриці за рядками є послідовність:

a[0][0], a[0][1],. .. a[0][n], a[1][0], a[1][1],. .., a[1][n], a[m][1], a[m][2],. .., a[m][n].

Тоді адреса будь-якого елемента визначається за формулою:

адреса a[i][j] = адреса a[0][0] + n*i + j;

Двовимірний масив можна розглядати як масив, елементами якого є стовпці матриці, тоді масив складається з m елементів. Упорядкування елементів масиву a за стовпцями є послідовність:

a[0][0], a[1][0], ... a[m][0], a[1][0], a[1][1], ..., a[m][2], a[1][n], a[2][n], ..., a[m][n].

Тоді адреса будь-якого елемента масиву визначається за формулою:

адреса a[i][j] = адреса a[0][0] + m*j + i;

9.3. МАСИВИ ВКАЗІВНИКІВ

Елементи масиву можуть мати будь-який тип, у тому числі й бути вказівниками. Масиви вказівників можна використати для роботи з усіма типами даних, але найчастіше їх використовують для зберігання символьних рядків різної довжини. Оголошення масиву вказівників виконується таким чином:

```
char *name[10]; // оголошення масиву вказівників name з 10 елементів
```

Елемент оголошення char* вказує, що тип кожного елемента масиву name — вказівник на char. Масиви вказівників можна ініціалізувати при оголошенні. Наприклад,

```
char *name [ ] = {"Понеділок", "Вівторок", "Середа", "Четвер",
"П'ятниця", "Субота", "Неділя"};
```

Компілятор резервує місце для семи вказівників. Вони набувають початкового значення, що дорівнює адресі початку в пам'яті відповідних рядків символів. На рис. 9.3 показано представлення масиву name в пам'яті.

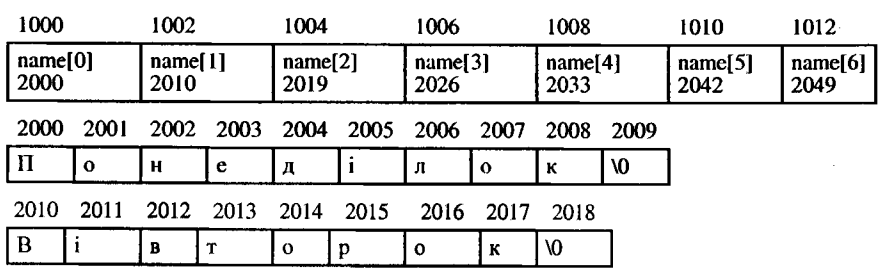


Рис. 9.3. Представлення в пам'яті масиву вказівників

Розділ 10

ЗАСОБИ РЕАЛІЗАЦІЇ ЦИКЛІЧНИХ ПРОЦЕСІВ

10.1. ОПЕРАТОР ЦИКЛУ З ВІДОМОЮ КІЛЬКІСТЮ ПОВТОРЕНЬ

До особливостей організації циклів з відомою кількістю повторень потрібно віднести обов'язкову наявність керуючої змінної, або ж параметра циклу, яка має певне початкове, кінцеве значення і закон визначення кожного наступного значення. Допоміжні дії, які виконуються з визначення початкового і поточного стану параметра циклу, а також з перевірки на вихід з циклу, називаються заголовком циклу. Заголовок використовується для організації повторень тіла циклу, тобто сукупності дій, що підлягають багаторазовому використанню.

Особливості організації заголовка циклу з відомою кількістю повторень реалізовані в операторі *for*. Формат оператора є таким:

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

де *вираз 1* задає початкове значення змінної, керуючої циклом, *вираз 2* визначає умову, при якій оператор циклу буде виконуватися, *вираз 3* визначає характер зміни керуючою змінною. Вирази в круглих дужках являють собою заголовок циклу, простий або складений оператор, узятий в фігурні дужки, являють тіло циклу. Схема виконання оператора є такою: спочатку обчислюється вираз 1, що зазвичай використовується для ініціалізації керуючої змінної циклом. Потім обчислюється вираз 2, якщо вираз істинний, виконується тіло циклу. Потім обчислюється нове значення керуючої змінної, задане виразом 3. Якщо вираз 2 істинний для нового значення керуючої змінної, то для цього значення керуючої змінної виконується тіло циклу. Процес продовжується доти, поки значення керуючої змінної не задовольнить умову виходу з циклу, визначену виразом 2, тобто вираз 2 стане хибним. Наприклад,

```
int i, s = 0;
for (i = 1; i < 10; i++)
   s+ = i;
cout << i;
```

Коли оператор циклу починає виконуватися, керуючій змінній *i* присвоюється початкове значення 1. Потім перевіряється умова

продовження циклу *i<10*. Оскільки початкове значення *i* дорівнює 1, то *i* менше 10, умова істинна, виконується оператор тіла циклу s+ = i;. Потім керуюча змінна збільшується на одиницю, тобто виконується *i++*. Далі перевіряється умова продовження циклу. Оскільки значення *i* дорівнює тепер 2, тобто є меншим від 10, то виконується тіло циклу. Цей процес продовжується, поки керуюча змінна *i* не збільшиться до 10, умова продовження циклу порушиться і повторення завершиться. Виконання програми продовжиться з першого оператора, розташованого після *for*, у цьому разі буде виконуватися оператор cout.

На рис. 10.1 подано опис оператора *for* у вигляді схеми алгоритму.

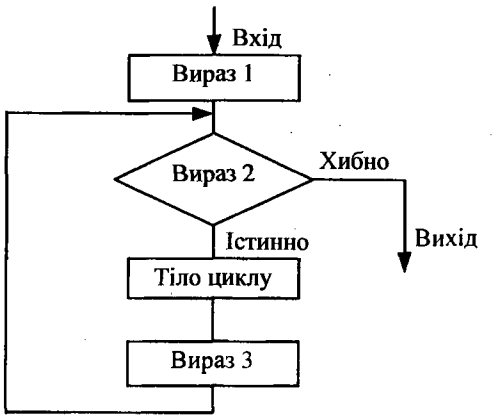


Рис. 10.1. Схема оператора for

Якщо в тілі *for* є більше за один оператор, то для визначення тіла циклу обов'язково потрібно взяти операторів у фігурні дужки. За синтаксисом мови C++ оператор може бути порожнім, у цьому разі в тілі циклу ставиться ; .

У мові C++ на відміну від інших мов програмування є широкі можливості для організації заголовка циклу оператора *for*. При перевірці гранично допустимого значення керуючої змінної циклом можна використати вираз, який формується на його основі. Наприклад,

```
for (k = 1; k + k < 100; k = k + 4)
;
```

що не припустимо в інших мовах програмування. У процесі зміни керуючої змінної у виразі C++ допустимі різноманітні форми запису, найхарактернішими з яких є використання опера-

цій +, –, ++, – –, *, /, +=, –=, *=, /=. Наприклад, як вираз С можуть використовуватися такі вирази: k = k + 2; k+ = 2; k– = 2; k++; k– –; k = k*3; k* = 3; k = k/2; k/ = 2; і т. п.

Мова С++ допускає опис змінної в самому заголовку функції, але область видимості змінних, які описані у виразі 1, розповсюджується лише на цикл *for*. У версіях, нижчих за Borland C++5.0, область видимості змінної визначалася від місця визначення до кінця блоку, що містить оператор *for*. Це означає, що визначені змінні в заголовку циклу використовуються всередині циклу. Тому під час компіляції програми, що містить наступний фрагмент, буде видана помилка.

```
int a[10], b[15];
for (int i = 0; i<10; i++)
   a[i] = 0;
for (i = 0; i<15; i++) // Компілятор видасть помилку Undefined
                       // symbol 'i'.
   b[i]=0;             // (Невизначений символ.)
```

Усунути цю помилку можна таким чином:

```
for (int i = 0; i<15; i++)
   b[i] = 0;
```

Кожен із трьох виразів може складатися з кількох виразів, об'єднаних оператором кома (,). Це дозволяє використати декілька змінних, що керують циклом.

```
for (step = 1, proc = 1; proc + step < = 50; step+ = 2, proc+ = 2)
```

Змінну циклу в операторі *for* можна порівнювати не з константою, а з іншою змінною. Наприклад,

```
int i, num, sum = 0, oz;
cin >> num;
for (i = 0; i< = num; i++)
   { cout << endl <<"Введіть оцінку"
     cin >> oz;
     sum + = oz; }
```

Будь-який з трьох виразів може містити будь-який коректний вираз, який може не виконувати дії, характерні для кожного виразу. Наприклад, на місці ініціалізації змінної циклу може стояти виклик бібліотечної функції або функції користувача.

```
for (printf ("\nВведіть число"); i = = 33; printf ("\nЧисло");
     scanf ("%d", & num);
```

Заголовок циклу містить виклик функцій *printf*, які виводять підказки для користувача, причому повідомлення "Введіть число" буде виведено один раз. Цикл завершиться, якщо буде введено число 33.

Змінні, що входять у вирази заголовка циклу, можна змінити під час виконання операцій у тілі циклу. Наприклад,

```
for (step = 1, proc = 1; proc< = 50; step* = 2)
    proc = proc + step + 2;
```

Вирази в заголовку *for* є необов'язковими, тому можна опустити будь-який з трьох виразів або все відразу, але при цьому потрібно залишити символ "крапка з комою". Якщо немає виразів 1 та 3, то керуюча змінна просто не використовується. Якщо пропущений вираз 2, то вважається, що він істинний і цикл не завершується. Таким чином,

```
for (; ; ) {...} або for (; 1;) {...}
```

є нескінченними циклами, з яких виходять іншими способами.

Вираз 1 у заголовку може бути винесений за межі оператора і замість нього можна записати будь-який оператор або функцію. Якщо вираз С++ відсутній, то зміна керуючої змінної повинна здійснюватися в тілі циклу, або зміна змінної не потрібна. У межах кожної частини заголовку виконання операторів присвоювання та операцій відношення здійснюються зліва направо.

Приклад використання оператора for. Скласти програму обчислення суми та кількості додатних та від'ємних елементів матриці.

```
#include <iostream.h>
#define m 10
#define n 10
void main()
{ int a[n][m];
  int sump = 0, sumo = 0, kolp = 0, kolo = 0, i, j, n1, m1;
  cout << endl << "Введіть розмірність матриці";
  cin >> m1 >> n1;
// введення матриці
for (i = 0; i < n1; i++)
 for (j = 0; j < m1; j++)
 { cout << endl << "Введіть a[" << i << "][" << j << "] ";
   cin >> a[i][j];
 }
// виведення матриці, що введена
for (i = 0; i < n1; i++)
{ cout << endl;
 for (j = 0; j < m1; j++)
    cout << a[i][j] << ' ';
}
// підрахунок додатних і від'ємних елементів
 for (i = 0; i < n1; i++)
  for (j = 0; j < m1; j++)
```

```
  { if (a[i][j] > 0)
{ kolp++;
 sump = sump + a[i][j];
}
 else
{ kolo++;
 sumo = sumo + a[i][j];
}
}
cout << endl << "Сума додатних елементів = " << sump;
cout << endl << "Кількість додатних елементів = " << kolp;
cout << endl << "Сума від'ємних елементів = " << sumo;
cout << endl << "Кількість від'ємних елементів = " << kolo;
  }
```

10.2. ІТЕРАЦІЙНІ ЦИКЛІЧНІ ПРОЦЕСИ

В ітераційних циклічних процесах неможливо заздалегідь передбачити кількість повторень. Перевірка на вихід з циклу здійснюється шляхом простої або складної умови. Доволі часто в ітераційних циклах відсутній параметр циклу, ці особливості повинні знайти відображення в заголовку відповідного оператора. У мові С++ існують два оператори, заголовки яких підходять для цих цілей. Формат першого з них є таким:

```
while <вираз>      // заголовок
<оператор>;        // тіло циклу
```

де вираз — це звичайно вираз типу порівняння, порожній, простий або складний, але може бути виразом будь-якого типу. Оператор — це порожній, простий або складний оператор.

Схема дії оператора така: спочатку обчислюється значення виразу, якщо обчислене значення не дорівнює нулеві, тобто вираз істинний, то виконується тіло циклу, потім вираз перевіряється знову. Ця послідовність дій виконується до тих пір, поки значення виразу не дорівнюватиме нулеві, тобто вираз прийме хибне значення. У цьому разі тіло циклу не виконується, а керування передається оператору, наступному за оператором циклу. Особливості використання оператора while:

1. Перед входженням в оператор мають бути визначені всі складові для обчислення;

2. У тілі циклу необхідно передбачити зміну параметрів для визначення кожного потокового значення виразу;

3. Якщо ітераційний процес має параметр, то переадресація параметра повинна відбуватися в тілі циклу;

4. Початкове входження у цикл можливе тільки в тому разі, коли початкове значення виразу не дорівнюватиме нулеві.

На рис. 10.2 подано опис оператора *while* у вигляді схеми алгоритму.

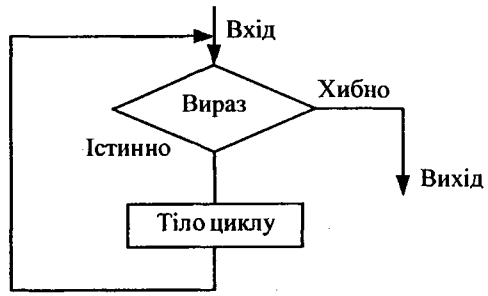


Рис. 10.2. Схема оператора while

Оператор *while* є оператором циклу з передумовою, оскільки істинність умови перевіряється перед входом у цикл, тобто тіло циклу може не виконатися жодного разу.

while (1); Даний оператор забезпечує нескінченний цикл з порожнім оператором як тіло циклу.

Приклад. Скласти програму обчислення наближеними методами значення ln(1 + x) за формулою $\ln(1+x)=\sum_{k=1}^{\infty}(-1)^{k+1}\cdot\frac{x}{k^k}$ з точністю ε, що вводиться з клавіатури. Вважати, що необхідна точність досягнута, якщо черговий доданок виявився за модулем меншим, ніж ε. Значення функції обчислити для x від 0 до 1 з кроком 0,1.

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
void main()
{
 int k; float x, e, r = 0, sum;
 cout << endl << "Введіть точність обчислення";
 cin >> e;
 for(x = 0; x< = 1.0; x+ = 0.1)
  { k = 1; sum = 0;
    r = pow(– 1, k + 1) (pow(x, k)/k;)
    sum+ = r;
```

```
    while (fabs(r) > e);
    {
      k++;
      r = pow(– 1, k + 1) (pow(x, k)/k;)
      sum+ = r;
    }
  cout << endl <<"ln("<< setprecision(1) << x+1 << ") ="
       << setprecision(5) << sum;
  }
}
```

Формат іншого оператора циклу для організації ітераційних процесів є таким:

```
do
<оператор>;
while(<вираз>);
```

де оператор тіла циклу може бути простим або складеним оператором, вираз — це будь-який допустимий у мові вираз.

Схема дії оператора така: спочатку, хоча б один раз, виконується тіло циклу; в цьому полягає його основна відмінність від оператора while. Після його завершення обчислюється вираз після ключового слова while. Якщо вираз істинний, то виконується тіло циклу, якщо ні, то здійснюється вихід з циклу, тобто керування передається наступному за ним операторові.

Особливості використання оператора є такими:

У тілі циклу необхідно передбачити зміну параметрів для обчислення виразу.

Якщо ітераційний процес має параметр, то його переадресацію слід здійснювати в тілі циклу.

На рис. 10.3. подано опис оператора *do while* у вигляді схеми алгоритму.

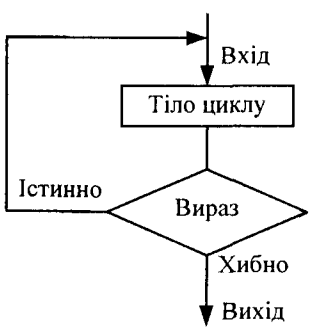


Рис. 10.3. Схема оператора do while

Оператор *do while* є оператором циклу з постумовою, тому тіло циклу виконується принаймні один раз. Нескінченний цикл з порожнім оператором як тіло можна записати таким чином:

```
do; while(1);
```

Приклад. Скласти програму знаходження ln(1+x), використовуючи оператор do. .. *while*

```
#include <iostream.h>
#include <iomanip.h>
#include <math.h>
void main ( )
{
 int k; float x, e, r = 0, sum;
 cout << endl << "Введіть точність обчислення";
 cin >> e;
 for (x = 0; x< = 1.0; x+ = 0.1)
  { k = 1; sum=0;
   do
   {
    r = pow(– 1, k+1)(pow(x, k)/k;)
    sum+ = r;
    k++;
   }
   while (fabs(r) > e);
  cout << endl << "ln("<< setprecision(1) << x << ") ="
       << setprecision(5) << sum;
  }
}
```

10.3. СКЛАДНІ ЦИКЛІЧНІ ПРОЦЕСИ

Складним називається циклічний процес, який містить один або кілька циклічних процесів аналогічного чи іншого вигляду. У програмі складний циклічний процес реалізується шляхом вкладання одного оператора циклу в тіло іншого оператора циклу. При цьому ніяких обмежень на вигляд оператора циклу не накладається, тобто зовнішній цикл може бути записаний за допомогою оператора *for*, а внутрішній може описуватися з допомогою *for, while* або *do-while* і навпаки.

У попередньому прикладі цикл *for* є зовнішнім циклом, а цикл *do-while* внутрішнім. Вкладеним називається цикл, що знаходиться всередині іншого циклу. Цикл *do-while* є вкладеним у цикл *for*.

10.4. ВИКОРИСТАННЯ ОПЕРАТОРІВ BREAK І CONTINUE В ОПЕРАТОРАХ ЦИКЛУ

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

```
#include < iostream.h>
void main ( )
{
 for (int i = 1; i < 10; i++)
 { if (i = = 7) // вихід з циклу при i = 7
   break;
 }
cout << endl << "При і рівному " << i << "цикл перерваний";
}
```

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

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

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

Розділ 11

КЛАСИ ПАМ'ЯТІ ТА ЇХ ВИКОРИСТАННЯ В МОДУЛЬНОМУ ПРОГРАМУВАННІ

11.1. КЛАСИ ПАМ'ЯТІ ТА ВИДИ ДІЙ ІМЕН ЗМІННИХ. ОБЛАСТЬ ВИДИМОСТІ ТА ЧАС ЖИТТЯ

Кожна змінна і функція, описана в програмі мовою С++, належить до якогось класу пам'яті. Клас пам'яті змінної визначає час її існування (час життя об'єкта), область видимості й місце, де об'єкт розташовується (внутрішні регістри процесора, сегмент даних, сегмент стека). *Область видимості* — це область початкового коду програми, з якого можливий коректний доступ до пам'яті або функції з використанням ідентифікатора. *Час життя* визначає час існування змінної в процесі виконання програми. З погляду часу життя об'єкта розрізнюють три типи об'єктів: статичні, локальні та динамічні.

Об'єкти зі *статичною тривалістю життя* отримують розподіл пам'яті на початку виконання програми — такий розподіл пам'яті зберігається до виходу з програми, розміщується в сегменті даних.

Об'єкти з *локальною тривалістю життя* створюються при вході в функцію, а при виході з функції знищуються, розміщуються в стеку чи регістрах.

Об'єкти з *динамічною тривалістю життя* створюються і знищуються спеціальними функціями керування пам'яттю, розміщуються в купі.

Існують чотири області видимості змінних: блок, функція, прототип функції і файл. Змінні, оголошені всередині блоку, мають областю дії блок. Блок починається оголошенням змінних і закінчується кінцевою правою фігурною дужкою блоку. Якщо блоки вкладені і змінна у зовнішньому блоці має таке саме ім'я, як змінна у внутрішньому блоці, змінна зовнішнього блоку невидима до моменту завершення роботи внутрішнього блоку. Змінні, оголошені на початку функції, областю видимості мають блок, який повністю містить тіло функції. Змінні, оголошені всередині прототипу функції, мають спеціальну область видимості, яка простягається від точки оголошення до кінця прототипу функції. Змінні, оголошені поза будь-якою функцією, мають областю дії файл.

Можливість роботи з різними класами пам'яті в процесі програмування дає програмісту механізм створення гнучких з погляду використання ОЗП програм. Клас пам'яті, що призначається змінній, визначає не лише тривалість збереження її значень, а й також спосіб інформаційного зв'язку з іншими функціями, які теж працюють з цією змінною. У мові С++ розрізняють чотири класи пам'яті, які означаються такими ключовими словами:

- auto (автоматична пам'ять);
- extern (зовнішня пам'ять);
- static (статична пам'ять);
- register (регістрова пам'ять).

Визначення класу пам'яті для змінної відбувається на етапі її оголошення в програмі. За замовчуванням, тобто якщо не задається програмістом, змінній призначається клас auto. Клас пам'яті для змінної задається або за розташуванням її опису, або за допомогою специфікаторів класу, перелічених вище. Специфікатори передують оголошенню змінної.

11.2. АВТОМАТИЧНІ ЗМІННІ

Автоматичні змінні, або ж *auto,* описуються всередині відповідної функції і ділянка їх дії лежить у межах даної функції, тобто змінна починає існування в момент активізації функції і зникає в момент завершення функції. Ще автоматичні змінні називають локальними. Спроба роботи з локальною змінною в інших функціях призводить до помилок. Оскільки за замовчуванням змінним призначається клас *auto,* то ключове слово *auto* можна не використовувати.

Два наведені нижче записи можна вважати еквівалентними.

```
float a, b;
auto float a, b;
```

Область видимості змінної автоматичного класу пам'яті починається з її визначення і завершується при виявленні кінця блоку, в якому ця змінна визначена. Доступ до таких змінних із зовнішнього блоку неможливий. Оскільки локальні змінні знищуються при виході з функції, в якій вони оголошені, то ці змінні не можуть зберігати значення між викликами функцій. Якщо не визначене місце для зберігання локальних змінних, то вони зберігаються в стеку. Пам'ять для автоматичних змінних відводиться динамічно під час виконання програми при вході в блок, у якому описана відповідна змінна. При виході з блоку пам'ять, відведена

під усі його автоматичні змінні, автоматично звільняється. Звідси й походить термін *автоматичні змінні.* Доступ до автоматичних змінних можливий лише з блоку, де змінні описані, оскільки до моменту входу в блок змінна взагалі не існує і пам'ять під неї не виділена.

11.3. ГЛОБАЛЬНІ ЗМІННІ

Зовнішні змінні, або ж *extern,* описуються поза функцією, але можуть бути описані і всередині функції — обов'язково зі специфікатором *extern.* Ділянкою дії змінних є всі функції програмного комплексу, тому ці змінні інакше називають *глобальними.*

Якщо змінна визначена перед основною функцією програмного комплексу, то в інших функціях, що входять у програмний файл, її можна не описувати як зовнішню, вона й так діятиме в них. До них можна отримати доступ у будь-якому виразі, незалежно від того, в якій функції знаходиться даний вираз.

Опис змінної як *extern* усередині функції потрібний у тих випадках, коли цю змінну необхідно використати, а вона визначена або в функції, яка активізується пізніше, або в іншому програмному файлі. Наприклад,

```
funk ( )
{ extern double pi;
...
}
double pi = 3.14159;
 void main ( )
{...
}
```

Функції main ( ) і funk ( ) розташовані в одному початковому файлі, але визначення зовнішньої змінної pi знаходиться після функції funk ( ), у цьому разі потрібне оголошення зовнішньої змінної pi в функції funk зі специфікатором extern. Оголошення зовнішніх змінних інформує компілятор, що така змінна вже існує і пам'ять для неї вже виділена.

Для того, щоб функції могли працювати з зовнішньою змінною, визначеною в іншому файлі, вони повинні також містити її оголошення. Наприклад,

```
// файл f1.cpp
double pi = 3.14159;
```

```
void main ( )
{...
}
// файл f2.cpp
funk ( )
{ extern double pi;
...
}
```

Якщо всередині блоку описана автоматична змінна, ім'я якої збігається з ім'ям глобальної змінної, то всередині блоку глобальна змінна маскується під локальну. Це означає, що всередині даного локального блоку буде видна автоматична змінна. Для розв'язання цієї проблеми в С++ можна використати операцію дозволу області видимості : :. Унарна операція : : дозволяє отримати доступ до зовнішньої іменованої області пам'яті для певної функції. Наприклад,

```
#include <iostream.h>
int x;              // Глобальна змінна
void main( )
{ int x=1234;  // Локальна змінна
  ::x=23;       // Присвоїти значення глобальній змінній
  cout << "\nЛокальна змінна x=" << x;
  cout << "\nГлобальна змінна x=" <<: :x;
}
```

Вираз : :x вказує, що використовується змінна x зовнішньої області дії, а не локальна.

Для зовнішніх змінних пам'ять відводиться один раз і залишається зайнятою до закінчення виконання програми. Якщо користувач не вкаже початкового значення глобальних змінних, то їм буде присвоєне значення нуль. Глобальні змінні зберігаються у фіксованій області пам'яті, що встановлюється компілятором.

11.4. СТАТИЧНІ ЗМІННІ

Статичні змінні, або ж *static*, мають таку ж ділянку дії, як і автоматичні змінні, але вони не зникають, коли функція, що їх містить, закінчить свою роботу. Компілятор зберігає їх значення від одного виклику функції до іншого. Локальна статична змінна при завершенні функції, в якій вона оголошена, не втрачає свого значення. Статична змінна залишається локальною

для цієї функції. При черговому виклику функції вона зберігає своє колишнє значення. Наприклад,

```
#include<iostream.h>
void main ( )
{int t = 0, i;
 int t (void);
 for(i = 0; i < 3; i++)
  { t = st( );
   cout << t;}
}
int st( )
{static int s = 0;
 s = s++;
 return(s);
}
```

Внаслідок роботи програми будуть виведені такі значення: 1, 2, 3. s ініціалізується 0 при першому виклику функції, при черговому виклику функції s не ініціалізується, вона зберігає попереднє значення.

Статичні змінні можуть бути описані поза будь-якою функцією, тоді вони інтерпретуються як зовнішні змінні. Різниця між зовнішньою змінною та статичною зовнішньою змінною полягає в тому, що остання з них може бути використана тільки функціями того програмного файла, в головній функції якого вона визначена, тобто доступна у власному файлі й більше ніде.

В С++ статична змінна, яка не ініціалізована, встановлюється в нуль.

Час життя статичних змінних глобальний: починається після визначення змінної і триває до кінця програми. Область видимості статичних змінних залежатиме від того, є вони зовнішніми чи внутрішніми.

11.5. РЕГІСТРОВІ ЗМІННІ

Регістрові змінні, або ж *register*, запам'ятовуються не в ОЗП, як усі інші змінні, а в регістрах центрального процесора, де доступ до них і робота з ними виконуються набагато швидше, ніж у пам'яті. В іншому регістрові змінні аналогічні автоматичним змінним.

При виділенні регістрової пам'яті запит програміста може бути не задоволений, оскільки регістри в поточний момент часу

можуть бути недоступні. У цьому разі регістрова змінна стає простою автоматичною змінною.

Змінні регістрового класу пам'яті мають такі самі область видимості та час життя, як і автоматичні змінні. Не можна застосовувати *register* до глобальних змінних.

У процесі написання програмних комплексів необхідно сполучати різні види змінних за використанням пам'яті. Найекономніше витрачається пам'ять при роботі з автоматичними змінними. Тому треба прагнути там, де можна, використовувати цей клас пам'яті, а інші застосовувати в тих випадках, коли це необхідно, а саме:

зовнішні змінні слід використовувати, коли потрібно організувати передачу великих обсягів даних між кількома функціями;

статичні змінні в тому разі, коли необхідно неодноразово звертатися до функції і в процесі кожного звертання використовувати результати попереднього звертання;

регістрові змінні використовувати з метою прискорення розрахунків.

У табл. 11.1 наведені область видимості та час життя для змінних різних класів пам'яті.

Таблиця 11.1

ОБЛАСТЬ ВИДИМОСТІ ТА ЧАС ЖИТТЯ ЗМІННИХ

Клас пам'яті	Специфікатор	Час життя	Область видимості
Автоматична	auto	Тимчасове	Локальна
Регістрова	register	Тимчасове	Локальна
Статична	static	Постійне	Локальна
Зовнішня	extern	Постійне	Глобальна (всі файли)
Зовнішня статична	static	Постійне	Глобальна (один файл)

11.6. ДИНАМІЧНИЙ РОЗПОДІЛ ПАМ'ЯТІ

У мові C++ програміст має можливість проводити розподіл пам'яті не лише на початку програми, а й у процесі її виконання в необхідному місці. Для цього слід указати розмір блоку пам'яті, що виділяється як аргумент бібліотечної функції malloc ( ). Прототип функції такий:

```
void *malloc(size_t size);
```


де size — необхідний розмір у байтах пам'яті, що виділяється, тип size_t визначений як беззнакове ціле.

Дана функція виділяє пам'ять і повертає вказівник типу void* на початок виділеної пам'яті, а якщо виділити пам'ять не вдається, то повертає NULL. Прототип функції знаходиться в заголовних файлах **<stdlib.h>** і **<alloc.h>**. Наприклад,

```
char *p;
p = (char*) malloc (100);
```

Виділяється область пам'яті в 100 байтів, адреса якої присвоюється змінній p. Оскільки malloc ( ) повертає вказівник типу void*, то необхідно використати при присвоєнні перетворення одного типу вказівника на інший.

Якщо необхідно виділити пам'ять для масиву, то це можна зробити за допомогою функції calloc ( ), прототип якої є таким:

```
void *calloc (size_t n, size_t size);
```

де n — кількість необхідних елементів пам'яті, size — розмір у байтах одного елемента, а розмір виділеної пам'яті дорівнює величині n*size. Повертає вказівник на перший байт виділеної пам'яті, якщо ж пам'яті недостатньо, то повертається нульовий вказівник. Наприклад,

```
float *p;
p = (float*) calloc (100, sizeof (float));
```

Виділяється пам'ять для 100 елементів, кожний розміром у 6 байтів.

Аргументи функцій malloc ( ) і calloc ( ) повинні бути цілого типу. Функція calloc ( ) додатково до виділення пам'яті здійснює обнулення всього виділеного блоку.

Для звільнення виділеного функціями malloc ( ) і calloc ( ) об'єму пам'яті використовується функція free ( ). Прототип функції є таким:

```
void free (void *ptr);
```

де ptr — вказівник на блок пам'яті, що звільняється.

Функція повертає пам'ять, на яку вказує параметр ptr, в купу. Для звільнення пам'яті з попереднього прикладу необхідно записати такий вираз:

```
free(p);
```

Оскільки пам'ять виділяється для певної мети і звільняється, коли її використання завершилося, то можна використати ту саму пам'ять в інший момент часу для інших цілей в іншій частині програми.

11.7. ОПЕРАЦІЇ NEW І DELETE ДЛЯ ДИНАМІЧНОГО РОЗПОДІЛУ ПАМ'ЯТІ

Мова С++ виділяє дві операції для динамічного розподілу пам'яті — new і delete. Операції *new* і *delete* виконують динамічний розподіл і скасування розподілу пам'яті, але з більш високим пріоритетом, ніж стандартні бібліотечні функції *malloc* і *free*. Формат їхній є таким:

<змінна вказівник> = new <тип змінної> (<ініціалізатор>);

delete(<змінна вказівник>);

де змінна вказівник — змінна типу вказівник; ініціалізатор — це вираз у круглих дужках.

Операція *new* дозволяє виділити і зробити доступною ділянку в основній пам'яті, розміри якої відповідають типу даних, що визначається ім'ям типу. У виділену ділянку заноситься значення, що визначається ініціалізатором, який не є обов'язковим елементом. У разі успішного виконання операції *new* повертає адресу початку виділеної ділянки пам'яті. Якщо ділянка потрібних розмірів не може бути виділена, то операція *new* повертає нульовий вказівник (NULL).

Вказівник, якому присвоюється значення адреси, що отримується, має відноситися до того самого типу даних, що й <тип змінної> в операції new. Наприклад,

```
new float;
new int(15);
```

У першому випадку операція *new* виділяє ділянку пам'яті розміром 4 байти, у другому — ділянку пам'яті в 2 байти й ініціалізує цю ділянку значенням 15.

```
int *h;              // визначення вказівника
h = new int(2);      // виділення пам'яті для int і ініціалізація 2
...
delete h;            // звільнення пам'яті
```

Вказівник h пов'язаний з ділянкою пам'яті, виділеною для величини цілого типу. Надалі доступ до виділеної ділянки пам'яті забезпечує вираз *h.

При виділенні пам'яті для масиву запис операції буде таким:

<змінна вказівник> = new <тип змінної> [<розмірність>];

Вертаний new вказівник, вказує на перший елемент масиву. Наприклад,

```
int *mat_ptr = new int[3][10];
```


Буде виділена ділянка динамічної пам'яті розміром 3*10*2 байти.

При виділенні динамічної пам'яті для масиву його розміри мають бути повністю визначені. Тільки перший (найлівіший) розмір масиву може бути заданий з допомогою змінної, інші розміри багатомірного масиву можуть бути визначені лише за допомогою констант.

Тривалість існування виділеної за допомогою операції *new* ділянки пам'яті — від точки її створення до кінця програми або до явного її звільнення за допомогою операції *delete*.

delete <змінна вказівник>;

де вказівник адресує ділянку пам'яті, що звільняється, раніше виділену за допомогою операції *new*. Повторне застосування *delete* до того самого вказівника дає невизначений результат.

Для звільнення динамічно розміщеного масиву необхідно використати таку форму запису *delete*:

delete[ ] <змінна вказівник>;

Квадратні дужки повідомляють про те, що необхідно звільнити пам'ять, виділену для масиву. Наприклад,

```
delete[ ] matr_ptr;
```

Приклад. Виділяється пам'ять для 10 елементів масиву типу float. Елементам масиву присвоюються значення від 100 до 109. Вміст масиву виводиться на екран.

```
#include <iostream.h>
void main()
{ float *p; int i;
  p = new float[10];
  for (i = 0; i < 10; i++)
    *(p + i) = 100 + i;
  for (i = 0; i < 10; i++)
    cout << *(p+i);
  delete [ ] p;
}
```

Операція *new* має ряд переваг перед функцією *malloc ( )*. По-перше, *new* автоматично обчислює розмір пам'яті, немає необхідності використати операцію *sizeof*. По-друге, операція *new* автоматично повертає вказівник необхідного типу, немає необхідності використовувати операцію перетворення типу. По-третє, є можливість ініціалізації об'єкта при використанні операції *new*.

Розділ 12

ОРГАНІЗАЦІЯ ФУНКЦІЙ У ПРОГРАМАХ І РЕАЛІЗАЦІЯ ЗВЕРНЕНЬ ДО НИХ

12.1. МОДУЛЬНА СТРУКТУРА ПРОГРАМ І СПОСОБИ ІНФОРМАЦІЙНОГО ЗВ'ЯЗКУ МОДУЛІВ

Програмне забезпечення автоматизованих комплексів задач має, як правило, модульну структуру, тобто складається з послідовності окремих модулів, упорядкованих за певною системою. У різних інструментальних системах програмування модуль постає у вигляді певної структурної одиниці: блоку, підпрограми, функції. У мові C++ модулем є функція.

Проектування програмного забезпечення здійснюється за заздалегідь вибраною проектувальником стратегією. Найчастіше використовується стратегія низхідного структурного проектування, під час якого розробка починається з визначення загальної функції, або ж мети, яка повинна бути реалізована комплексом загалом, а потім реалізовується покрокова деталізація цієї функції доти, доки весь комплекс не буде описаний достатньою мірою докладно.

На кожному кроці одну чи кілька окремих підцілей функцій представляють або за допомогою окремих модулів, або груп модулів. При цьому з'являється необхідність у деталізуванні нових функцій і т. д.

Зрештою, загальна функція програмного комплексу виражається за допомогою еквівалентної структури відповідним чином сполучених підфункцій. Кожна з таких підфункцій вирішує лише частину задачі, однак вона є простішою від початкової функції і часто допускає подальше розбиття, об'єднання та перевірку. Описаний процес не повинен бути чисто формальним. Якщо отримана на якомусь кроці функція достатньо проста, то її не слід штучно ускладнювати шляхом подальшого розбиття.

Після того, як визначена функціональна структура програмного комплексу, з'ясовується технологія обробки даних, тобто здійснюється вибір пакетної або діалогової технології. У теперішній час найчастіше використовується діалогова технологія, при якій доступ користувача до процесу обробки даних різко розширюється, тобто користувач отримує можливість звертатися послідовно до модулів, що обробляють кожен крок. Інтерфейс користувача, тобто його зв'язок з процесом обробки даних, у цьому варіанті доцільно реалізувати на основі системи меню різного рівня ієрархії.

Програмний комплекс мовою C++ являє собою множину функцій, між якими реалізуються визначені інформаційні зв'язки і які активізуються в наперед визначеній та ієрархічно упорядкованій послідовності. Після визначення модульної структури програмного комплексу програміст вирішує питання про структуру вхідних та вихідних даних, які мають бути строго уніфіковані в межах задач, про склад робочих змінних та класи використовуваної пам'яті. Для вибору класу пам'яті для кожної конкретної змінної аналізуються можливості передавання її значення в інші модулі, вимоги до збереження її значення в момент завершення певного модуля.

Область дії змінних вибирається в межах використовуваного програмістом класу пам'яті, визначає вид інформаційного зв'язку між окремими модулями. Якщо програма використовує автоматичні чи реєстрові змінні, то міжмодульний зв'язок повинен бути організований через аргументи і параметри, які безпосередньо вказуються у процесі звертання до функції і в заголовковій її частині. У цьому разі йдеться про використання функції з параметрами.

При використанні вказаних вище видів змінних може виникнути ситуація, коли необхідно активізувати функцію без передавання аргументів, тобто між модулями не існує інформаційного зв'язку. В цьому випадку функція оформлюється як функція без параметрів і в момент її активізації список аргументів не описується. Якщо програміст використовує зовнішні змінні, то інформаційний зв'язок між модулями здійснюється на їх основі без передавання аргументів та параметрів. У цьому разі говорять про використання функції без параметрів.

12.2. ВИЗНАЧЕННЯ, ОГОЛОШЕННЯ ТА ВИКЛИК ФУНКЦІЇ

Функція — це логічно самостійна частина програми, котрою можуть передаватися параметри і яка може повертати значення. Кожна функція повинна мати ім'я для виклику функції. Ім'я однієї з функцій — *main( )* — має бути присутнім у кожній програмі, і воно зарезервоване.

При роботі з локальними змінними інформаційний зв'язок є єдиним видом міжмодульного зв'язку. З використанням функцій у мові С++ пов'язані три поняття: визначення функції; оголошення функції; виклик.

Визначенням функції називається код, який описує те, що робить функція. Визначення функції потрібно проводити за такою схемою:

<тип вертаного значення> <ім'я функції> (<список параметрів>)
{<тип> <внутрішні змінні>;
<оператори;
[return(вираз);]
}

Перед першою фігурною дужкою стоїть заголовок визначення функції, між фігурними дужками міститься тіло визначення функції. *Тіло функції* — це складовий оператор, що містить оператори, які визначають дії функції. Синтаксис мови С++ забороняє всередині визначення функції поміщати визначення ще однієї функції. Тексти всіх функцій повинні бути записані послідовно, один за іншим.

```
void fun (int x, float y) // заголовок функції
{                         // початок тіла функції
...
}                         // кінець тіла функції
```

Тип вертаного функцією значення вказує тип даних результату, що повертається із функції оператору її виклику. В С, якщо тип значення, що повертається функцією, не заданий, то функція за замовчуванням повертає значення int. В С++ необхідно явно оголошувати тип значення, що повертається функціями. Якщо функція не вертає значення, то її тип повинен бути вказаний як void.

Ім'я функції формується за правилами написання ідентифікатора. Потрібно пам'ятати, що синтаксис мови С++ чутливий до регістра символів, тому

```
int Fun( )
int fun( )
int FUN( )
```

відносяться до заголовків трьох різних функцій.

Список параметрів — інакше його називають списком формальних параметрів — розділений комами список змінних, які приймають значення в момент активізації функції під час її виклику. Як об'єкти списку можуть використовуватися прості змінні, масиви, вказівники, а також типи, що визначаються користувачем. Список параметрів може бути відсутнім, тоді список параметрів задається як *void*. Список параметрів береться в круглі дужки, після яких точка з комою не ставиться. С++ відрізняється від С способом завдання порожнього списку параметрів. У С порожній список параметрів означає, що перевірка аргументів відсутня, при виклику функції може бути переданий будь-який аргумент. У С++ це означає відсутність аргументів, наступні два оголошення еквівалентні. Наприклад,

```
int fun ( );
int fun (void);
```

Усі об'єкти списку мають бути описані відповідними типами безпосередньо в списку параметрів перед відповідним елементом списку.

```
float Sum (float v, int d)
```

Потім записується складний оператор, у якому описуються змінні, що беруть участь у розрахунках усередині функції і за необхідності оператор повернення значення *return*. Оператор *return* може повертати в точку виклику функції лише одне значення, яке може бути виразом, змінною, константою чи вказівником. Крім передачі значень оператор *return* завершує виконання функції і передає керування наступному оператору викличної функції, навіть у тому разі, якщо він не є останнім. Коли функція не повертає ніякого значення, то вираз в операторі відсутній, у цьому випадку оператор *return* можна опустити.

Звертання до функції має бути реалізоване з іншої функції, в якій обов'язково повинен міститися опис типів вертаного функцією значення і бути вказане ім'я функції зі списком аргументів (або ж фактичних аргументів чи фактичних параметрів) у потрібному місці будь-якого виразу.

<ім'я функції> (<список аргументів>);

Аргументи повинні бути попередньо описані. У списку аргументів вони знаходяться у тій самій послідовності, що й у списку параметрів функції. Аргументи передаються з викличної функції в функцію, що викликається, за значенням, тобто обчислюється значення кожного виразу, що представляє аргумент, і це значення використовується в тілі функції замість відповідного формального параметра.

Аргументи можуть бути константами, простими змінними, масивами, вказівниками. Безпосередньо в момент активізації функції відбувається формування стека передаваних у функцію

значень, в якому довжина кожного значення визначається типом аргументу, що передається. Виняток становлять вказівники, при їх використанні відбувається не передавання значень, а переадресація параметрів функції за конкретними адресами пам'яті.

Оголошення функції — це оператор, що включає в себе тип вертаного значення, ім'я функції та її параметри. Цей оператор закінчується крапкою з комою. Мова С++ вимагає, щоб оголошення функції передувало її визначенню або першому використанню в програмі. Таке оголошення називається прототипом функції. Прототип функції вказує компілятору тип даних, що повертаються функцією, кількість параметрів, яку чекає функція, тип параметрів і очікуваний порядок їх слідування. Компілятор використовує прототип функції для перевірки правильності викликів функції. Прототип функції має вигляд:

<тип вертаного значення> <ім'я функції> (<список аргументів>);

Наприклад, int sum (int, int, int);

Цей прототип вказує, що sum має три аргументи типу int і повертає результат типу int.

Прототип розміщується у викличній програмі до заголовку *main( )*. При наявності прототипу передбачається перетворення аргументів до активізації функції в тип, завданий для відповідності параметрів у прототипі. Наприклад,

```
int f(float a, float b)
main ( )
{ float a1, int b1;
  t = f(a1, b1);
  . . . }
```

Невідповідність типів аргументів та параметрів може призвести до неправильної інтерпретації передаваних значень у момент активізації функції, коли формується стек значень, передаваних у функцію.

Прототип функції, розміщений поза описом якоїсь функції, відноситься до всіх викликів даної функції, які виникають після цього прототипу в даному файлі. Прототип функції, розміщений всередині опису певної функції, відноситься лише до викликів усередині цієї функції.

Другим способом оголошення функції перед використанням є поміщення прототипів функцій у заголовний файл (файл з розширенням .h), який підключається директивою #include до тексту програми.

Списки аргументів і параметрів можуть містити невизначену кількість об'єктів. У цьому разі у списку параметрів після останнього ставляться крапки (. . .). Якщо у списку параметрів вказані тільки крапки, то список може бути порожнім.

Прототип функції, заголовок функції і виклик функції повинні бути узгоджені між собою за кількістю, типом, порядком слідування аргументів і параметрів та за типом вертаних значень.

Наприклад, визначена користувачем функція sum( ) використовується для розрахунку суми ряду. Ця сума вертається з допомогою оператора return у функцію main( ) і виводиться на друк.

```
// Підрахунок суми ряду
#include<iostream.h>
int sum(int, int, int); // прототип функції
main()
{ int nach, kon, shag;
  cout << "\n Введіть початкове значення члена ряду: ";
  cin >> nach;
  cout << "\n Введіть кінцеве значення члена ряду: ";
  cin >> kon;
  cout << "\n Введіть крок зміни члена ряду: ";
  cin >> shag;
  cout << "\n Сума ряду дорівнює " << sum (nach, kon, shag);
   return 0;
}
// Визначення функції sum
int sum (int nachr, int konr, int shagr)
{ int i, sumr = 0;
  for(i = nachr; i< = konr; i+ = shagr)
     sumr+ = i;
  return (sumr);
}
```

12.3. ОРГАНІЗАЦІЯ ТА АКТИВІЗАЦІЯ ФУНКЦІЙ З ІНФОРМАЦІЙНИМ ЗВ'ЯЗКОМ ЧЕРЕЗ АРГУМЕНТИ І ПАРАМЕТРИ. ПЕРЕДАЧА ЗНАЧЕНЬ ФУНКЦІЇ

В С++ три способи передачі аргументів у функцію: передача за значенням, передача за посиланням з аргументами-вказівниками і передача за посиланням з аргументами-посиланнями.

Коли аргументи передаються в функцію за значенням, то відбувається створення копій аргументів, передача їх у функцію і присвоєння параметрам. Викликана функція працює з копією ар-

гументів, тому жодні зміни значень параметрів не відіб'ються на зміні аргументів. Якщо за значенням передається більше, ніж один аргумент, то копії кожної з них присвоюються відповідним параметрам функції, яку викликають.

У попередньому прикладі функція sum( ) передає копії змінних nach, kon, shagr, які використовуються функцією як значення для параметрів nachr, konr, shagr.

Виклик функції за значенням застосовується в тих випадках, коли обсяг аргументів, що передаються в функцію, невеликий і функція не повертає великого об'єму даних.

12.4. ВИКОРИСТАННЯ ВКАЗІВНИКІВ ПРИ РОБОТІ З ФУНКЦІЄЮ З ДАНИМИ АРГУМЕНТАМИ І ПАРАМЕТРАМИ. ПЕРЕДАЧА ВКАЗІВНИКІВ

Щоб мати можливість безпосередньо змінювати значення змінних викличної програми, необхідно використати передачу за посиланням з аргументами-вказівниками. Це, звичайно, необхідне в разі, коли в викличну програму потрібно передавати більше одного значення і повернути більше за одне значення. При цьому передаються не копії змінних, а копії адрес змінних. Функція, використовуючи вказівник, здійснює доступ до потрібних елементів пам'яті, і можна змінювати значення об'єкта, розташовані за цією адресою. Дана передача дозволяє передавати в викличну функцію масиви як аргументи. Якщо масив використовується як аргумент функції, передається тільки адреса масиву, а не копії всіх елементів. При виклику функції з ім'ям масиву в функцію передається вказівник на перший елемент масиву. Існує кілька способів оголошення параметра, призначеного для отримання вказівника на масив.

Перший варіант полягає в тому, щоб використати як аргумент функції ім'я масиву. При передачі масиву в функцію передається також розмірність, щоб функція могла обробляти задане число елементів у масиві. Наявність розмірності індексів в оголошенні параметра повідомляє функцію, як розташовані елементи в масиві. Якщо ми передаємо одновимірний масив як аргумент функції, в списку параметрів функції розмірність може бути опущена, тобто квадратні дужки будуть порожніми. Розмірність першого індексу багатомірного масиву також можна не вказувати, але вся інша розмірність індексів повинна бути зазначена. Наприклад, підрахувати питому вагу додатних і непарних за значенням елементів.


```
// Демонстрація передачі в функцію двовимірного масиву
#include <stdio.h>
#define n 10
float nd (float [ ][n], int);
 void main( )
{
 float a[n][n], aa;
 int i, j, nn;
 printf ("\nВведіть кількість рядків");
 scanf (" %d", &nn);
  for (i = 0; i<nn; i++)
    for (j = 0; j<nn; j++)
    {
     printf ("\nВведіть елемент");
     scanf ("%e", &aa);
     a[i][j] = aa;
    }
 printf ("\n Результат %f", nd(a, nn));
}
#include <math.h>
float nd (float a1[][10], int n1)
{
 int i, j;
 float s1 = 0, s2 = 0;
  for (i = 0; i<n1; i++)
    for (j = 0; j<n1; j++)
    {
    s1+ = a1[i][j];
    if(a1[i][j]>0 && fmod(a1[i][j], 2)! = 0)
       s2+ = a1[i][j];
    }
return (s2/s1);
}
```

Другий спосіб полягає в тому, щоб використати як аргумент функції вказівник на масив. Проілюструємо це на такому прикладі. Парні за значенням елементи матриці піднести в квадрат, а непарні зменшити в два рази. Підрахувати питому вагу парних і непарних елементів.

```
#include <stdio.h>
#define n 5
#define m 5
void nd (float*, double*, double*, double*, double*, double*, int, int);
```

```
main( )
{
 float a[n][m], aa;
 double s1 = 0, s2 = 0, s3 = 0;
 double nd1, nd2;
 int nn, mn, i, j;
 float *ra;
 ra = &a[0][0];
 printf ("\nВведіть кількість рядків ");
 scanf (" %d", &nn);
 printf ("\nВведіть кількість стовпців ");
 scanf (" %d", &mn);
  for (i = 0; i<nn; i++)
    for (j = 0; j<mn; j++)
    {
    printf ("\nВведіть елемент матриці");
    scanf (" %e", &aa);
    a[i][j] = aa;
    }
 nd (ra,&s1,&s2,&s3,&nd1,&nd2, nn, mn);
 printf ("\n результат %f і %f", nd1, nd2);
}
#include <math.h>
void nd (float *rra, double *rs1, double *rs2, double *rs3, double *rnd1,
          double *rnd2, int n1, int m1)
{
 int i, float k;
  for (i = 0; i<n1*m1; i++)
    {
    k = *(rra+i);
    if (fmod(k, 2) = = 0)
    {
    *(rra+i)* = *(rra+i);
    *rs1+ = *(rra+i);
    }
    if (fmod (k, 2)! = 0)
    {
    *(rra+i)/ = 2;
    *rs2+ = *(rra+i);
    }
 *rs3+ = *(rra+i);
 }
 *rnd1 = (*rs1)/(*rs3);
 *rnd2 = (*rs2)/(*rs3);
 return;
 }
```


12.5. ПЕРЕДАЧА ЗА ПОСИЛАННЯМ

В C++ можна здійснювати передачу за посиланням з аргументами-посиланнями. *Посилання* є неявним вказівником і використовується як інше ім'я вже існуючого об'єкта. Формат визначення посилання є таким:

<тип>& <ім'я посилання> = <вираз; або
<тип>& <ім'я посилання> (<вираз>);

Як вираз, що ініціалізується, має бути ім'я певного об'єкта, що має місце в пам'яті. Значенням посилання після визначення з ініціалізацією стає адреса цього об'єкта. Наприклад,

```
int x = 7;          //Визначена й ініціалізована змінна x
int& ref = x;       //Визначене посилання ра і значенням її є ад-
                      реса змінної x
```

Для доступу до вмісту ділянки пам'яті, на який вказує посилання, не треба виконувати розіменування, що є обов'язковим при зверненні до значення змінної через вказівник. Кожна операція над посиланням є операцією над тим об'єктом, з яким вона пов'язана. Можна змінити значення x, записавши x = 35; або ref = 35;.

Існують обмеження при визначенні і використанні посилань:

1. Не можна взяти адресу змінної типу посилань;
2. Не можна створити масив посилань;
3. Не можна створити вказівник на посилання;
4. Не допускаються посилання на бітові поля.

При використанні посилання як параметра забезпечується доступ із тіла функції до відповідного аргументу, тобто ділянки пам'яті, виділеної для аргументу, тобто початкова змінна може бути змінена за допомогою викличної функції. Щоб указати, що параметр функції передається за посиланням, після типу параметра в прототипі функції записується символ &. У виклику функції змінна вказується на ім'я, але вона буде передана за посиланням. Класичним прикладом передачі аргументів за посиланням є функція, що міняє місцями значення двох своїх аргументів.

```
#include <iostream.h>
void ref(int&, int&);
void main()
{ int a = 7, b = 17;
  ref(a, b);
  cout << "a=" << a << endl << "b=" << b;
}
void ref(int& x, int& y);
 { int z;
   z = x;
   x = y;
   y = z;
}
```

12.6. ОРГАНІЗАЦІЯ ТА АКТИВІЗАЦІЯ ФУНКЦІЙ З ІНФОРМАЦІЙНИМ ЗВ'ЯЗКОМ ЧЕРЕЗ ЗОВНІШНІ ЗМІННІ

Областю дії зовнішніх змінних є всі функції програмного комплексу, якщо ці змінні описані перед якою-небудь функцією, або як *extern* усередині функції. Це означає, що працювати з зовнішніми змінними можна в межах комплексу і значення, що формуються при цьому, доступні для всіх функцій. У цьому разі немає необхідності здійснювати міжмодульний зв'язок через аргументи і параметри, а достатньо лише активізувати необхідну функцію і продовжити виконання дії з зовнішніми змінними, які були розпочаті в інших функціях. Імена змінних в усіх функціях для позначення відповідних об'єктів повинні обов'язково бути однаковими.

Приклад. Визначити суму додатних і від'ємних елементів матриці.

```
#include <iostream.h>
const int n = 10, m = 10;
int supm, sumo;
int main( )
{ int a[n][m], *pa, koli, kolj;
  void sum(int *, int, int);
  pa = &a[0][0];
  cout << "\n Введіть розмірність матриці";
  cin >> koli >> kolj;
  sum(pa, koli, kolj);
  cout << endl << "Сума додатних елементів дорівнює" << sump
               << endl << "Сума від'ємних елементів дорівнює"
               << sumo;
}
void sum (int *ppa, int koli, int kolj))
{ extern int sump = supo = 0; int i, j;
  for (i = 0; i<koli*kolj; i++)
  { cout << endl <<"Введіть елемент матриці";
    cin >>*(ppa+i);
    if (*(ppa+i )> = 0)
      sump+ = *(ppa+i);
    else
      sump+=*(ppa+i);
  }
}
```


12.7. РЕКУРСИВНІ ФУНКЦІЇ

Функція називається *рекурсивною,* якщо вона викликає сама себе. Розрізняють пряму та непряму рекурсії. Функція називається непрямою рекурсивною в тому разі, якщо вона містить звернення до іншої функції, що містить прямий або непрямий виклик першої функції. Якщо в тілі функції явно використовується виклик цієї функції, то має місце пряма рекурсія. Класичним прикладом рекурсії є обчислення факторіала. Факторіал невід'ємного цілого числа n дорівнює

n x (n–1) x (n–2) x. .. x 3 x 2 x 1,

причому за визначенням факторіал 1! і 0! дорівнює 1. Факторіал числа дорівнює добутку попередніх щодо нього послідовностей чисел, тобто n! = nx(n–1)!. Якщо обчислення факторіала оформити у виді рекурсивної функції, то програма може бути подана в ось якому вигляді:

```
// Обчислення факторіала
#include <iostream.h>
unsigned long factorial (unsigned int);
void main( )
{ int i; unsigned int n;
  cout << endl << "Введіть ціле додатне число";
  cin >> n;
  for (i = 0; i< = n; i++)
      cout << endl << " Факторіал " << i << "!=" << factorial(i);
  return;
}
unsigned long factorial (unsigned int num);
{if (n = = 1 || n = = 0)
      return 1;
else
      return (num*factorial(num-1));
}
```

Рекурсивні функції обробляються повільно й займають більше стекової пам'яті, ніж їхні нерекурсивні еквіваленти. Надто велика кількість рекурсивних викликів функції може призвести до переповнення стека. Оскільки місцем зберігання параметрів і локальних змінних функції є стек, і кожен новий виклик створює нову копію змінних, простір стека може бути вичерпаний, це викличе помилку переповнення і призведе до аварійної зупинки програми.

12.8. ВБУДОВАНІ ФУНКЦІЇ

Вбудовані функції (inline) — це функції, чиє тіло підставляється в кожну точку виклику, замість того, щоб генерувати код виклику. Модифікатор *inline* перед вказівкою типу результату в оголошенні функції загадує компілятору згенерувати копію коду функції у відповідному місці, щоб уникнути виклику цієї функції. У результаті виходить множина копій коду функцій, вставлених у програму, замість єдиної копії, якою передається керування при кожному виклику функції. Модифікатор *inline* необхідно застосовувати для невеликих і часто використовуваних функцій. Наприклад,

```
#include <iostream.h>
inline int min (int x1, int y1)
{ if (x1 < y1)
    return (x1);
  else
    return (y1);
}
void main( )
{ int x, y, z;
{ cout << endl << "Введіть два цілих числа ";
  cin >> x >> y;
  z = min(x, y);
  cout << endl << " Мінімальним з " << x << " і " << y << " є "
       << z;
}
```

Компілятор може ігнорувати модифікатор *inline*, якщо вбудований код надто великий, компілятор згенерує звичайний виклик функції.

12.9. ПЕРЕВАНТАЖЕНІ ФУНКЦІЇ

C++ дозволяє визначати функції з однаковими іменами, але унікальними типами аргументів. У цьому разі про функції кажуть, що вони перевантажені. Перевантаження функції використовується звичайно для створення кількох функцій з однаковим ім'ям, призначених для виконання схожих задач, але з різними типами даних. Наприклад,


```
#include <iostream.h>
int fun (int x, int y)
{ return (x*x+y*y);}
float fun (float x, float y)
{ return (x*x+y*y); }
main( )
{ int x1 = 10, y2 = 3; float x2 = 13.75, y2 = 11.25
cout << endl <<"Сума квадратів чисел " << x1<< " і " << y1
     << "дорівнює "
     << fun(x1, y1)
     << endl <<"Сума квадратів чисел " << x2<< " і " << y2
     << "дорівнює "
     << fun(x2, y2)
return 0;
}
```

Перевантажені функції розрізняються за допомогою сигнатури — комбінації імені функції і типів її параметрів. Компілятор кодує ідентифікатор кожної функції за числом і типом її параметрів, це називається декоруванням імені, щоб мати можливість здійснювати надійне зв'язування типів. Надійне зв'язування типів гарантує, що викликається належна функція і що аргументи узгоджуються з параметрами. Компілятор виявляє помилки зв'язування і видає повідомлення про них. Кожне декороване ім'я починається з символу @, який передує імені функції. Закодований список параметрів починається з символів $g. Типи значень функцій, що повертаються, не відбиваються в декорованих іменах. Перевантажені функції можуть мати різні або однакові типи значень, що повертаються, і обов'язково повинні мати різні списки параметрів. Дві функції, відмінні лише за типами значень, що повертаються, викличуть помилку компіляції.

Розділ 13

СТРУКТУРА ДАНИХ

13.1. ПОНЯТТЯ СТРУКТУРИ

Мова C++ володіє могутніми засобами представлення складних даних і надає користувачеві можливість конструювати з основних типів похідні типи даних: структури даних, бітові поля, об'єднання, перелічувальний тип і створення нових типів. У цьому розділі розглянемо особливості структур.

Структура — це тип складових (агрегатованих) даних, що являють собою сукупність різноманітних елементів, логічно пов'язаних між собою в процесі розв'язання задачі. Прикладом структури може служити особиста картка студента. Студент описується набором атрибутів, таких, як номер залікової книжки, ім'я, дата народження, стать, адреса, факультет, курс, група і т. д. В свою чергу, деякі з цих атрибутів самі можуть виявитися структурами, наприклад, ім'я, дата народження, адреса. Кожна структура включає в себе один чи кілька об'єктів (змінних, масивів, вказівників, структур), які називаються елементами структури. Елементи структури часто називають членами структури або полями. Структура може мати ім'я, і кожен елемент структури обов'язково повинен мати унікальне в межах структури ім'я. Елементи, об'єднані у структуру, можуть бути різного типу й обов'язково описуватися відповідними атрибутами.

13.2. ШАБЛОН СТРУКТУРИ. СТРУКТУРНА ЗМІННА

Термін «структура» в мові відповідає двом різним поняттям:

1) позначення місця у пам'яті, де розміщується інформація (далі це місце називається *структурною змінною*);

2) правила формування структурної змінної, використовуваної компілятором для виділення їй місця в пам'яті та організації доступу до її полів. Далі такі правила називаються *шаблоном структури* або структурною змінною.

Як і будь-яка змінна, структурна змінна повинна описуватися в програмі. Опис структурної змінної у програмі відбувається в два етапи:

1) опис структурного шаблону;

2) означення структурної змінної.

Формат опису структурного шаблону є таким:

```
struct [<ім'я шаблону>]
    { <тип елемента><ім'я елемента>[розмірність];
...
    };
```

де *struct* — ключове слово поняття «структура», яке повідомляє компілятору про оголошення структури;

<ім'я шаблону> інакше «тег», ім'я типу структури або ярлик структури є необов'язковим і може формуватися довільно за правилами побудови ідентифікаторів мови. Ім'я шаблону не є ім'ям змінної, воно являє собою мітку, яку можна використати для посилання на цю структуру. Ім'я шаблону є обов'язковим у тому разі, коли шаблон визначений в одному місці, а фактичні змінні в іншому. Для імені шаблону пам'ять не резервується, поки не буде оголошена відповідна структурна змінна. Імена шаблонів повинні бути унікальними в межах їх області визначення. У межах однієї функції може бути лише один шаблон без імені. Ім'я шаблону може збігатися з іменами елементів шаблону, іменами змінних або міток.

Далі перераховуються через крапку з комою описи елементів, що об'єднуються в структуру. Для кожного елемента задається тип, ім'я і, якщо елемент є масивом, то і його розмірність. Імена елементів в одному шаблоні повинні бути унікальними. У різних шаблонах можна використати імена елементів, що збігаються.

Оголошення завершується крапкою з комою, оскільки оголошення структури — це оператор.

Шаблон структури є лише описом схеми розміщення її елементів, яка повідомляє компілятору, але не викликає ніяких дій, тобто визначена тільки форма даних. Задання шаблону не пов'язано з резервуванням будь-якої пам'яті компілятора. Шаблон дає компілятору всю необхідну інформацію про поля структурної змінної для резервування пам'яті й організації доступу до цієї пам'яті під час опису структурної змінної і посилання на окремі поля структурної змінної.

Означення структурної змінної полягає у накладенні шаблону на реальну змінну пам'яті. Довжина змінної, що виділяється під структуру, дорівнює сумі довжин усіх її елементів.

Означення структурної змінної можна здійснювати двома способами. В першому випадку ім'я змінної записується безпосередньо після шаблону, тобто після дужки }, що закривається, вказується ім'я змінної і ставиться крапка з комою ; . При цьому ім'я типу структури вказувати не потрібно, бо опис шаблону суміщений з визначенням структурної змінної. Вигляд оголошення структури тоді є таким:

```
struct [<ім'я шаблону>]
    { <тип елемента><ім'я елемента>[розмірність];
...
    }< список імен структурних змінних>;
```

У цьому випадку визначення структурного шаблону та структурної змінної суміщені. Імена структурних змінних у списку відокремлюються між собою комами. Структурні змінні або ім'я шаблону можуть бути відсутніми, але не одночасно. Наприклад,

```
struct anketa
  { int zalik_nom;
    char fio[20];
    char pol;
  } st1, st2;
```

Шаблон має ім'я anketa, структура містить три елементи: ціле, символьний масив і символьне. Імена елементів: zalik_nom, fio, pol; оголошені дві структурні змінні st1, st2, для яких компілятор резервує пам'ять.

У іншому випадку потрібна обов'язкова наявність імені типу структури (імені шаблону), оскільки визначення структурної змінної виноситься за опис шаблону. Формат визначення змінної при цьому є таким:

```
struct <ім'я шаблону> <ім'я структурної змінної>;
```

Другий варіант визначення зручно використати в тому випадку, коли структурний шаблон використовується більше за один раз. Наприклад,

```
#include <stdio.h>
#define size1 30
#define size2 6
      struct stt
      { char fio[size1];    // ПІБ студента
        char nz[size2];     // номер залікової книжки
        float sum_st;       // сума стипендії
      };
void main( )
struct stt st;              // визначення структурної змінної
```


13.3. ІНІЦІАЛІЗАЦІЯ СТРУКТУР

Існує два способи ініціалізації елементів структури. Можна проініціалізувати елементи при оголошенні структури або виконати ініціалізацію в тілі програми.

Ініціалізація структурних змінних припускається лише в тому разі, коли вони описані як зовнішні чи статичні. Наприклад,

```
#include <stdio.h>
#define size1 30
#define size2 6
      struct stt
      { char fio[size1];
        char nz[size2];
        float sum;
      };
struct stt st =          //оголошення й ініціалізація змінної st типу stt
{"Іваненко",
 "608103",
  25.00};
```

Дозволяється об'єднувати задання шаблону, опис структурних змінних і їх ініціалізацію в одному реченні мови С++, наприклад,

```
#include <stdio.h>
#define size1 30
#define size2 6
      struct stt
      { char fio[size1];
        char nz[size2];
        float sum;
      }st ={
"Іваненко",
"608103,
 25.00};
```

Іншим способом є ініціалізація окремих елементів структурної змінної в тілі програми за допомогою операції крапка '.'. Наприклад, st.nz = "608103";. За допомогою даної операції можна вводити дані в структуру з клавіатури, використовуючи функції введення.

13.4. ЗОВНІШНІЙ ТА ВНУТРІШНІЙ ШАБЛОНИ

Як і проста змінна, шаблон має область визначення (видимість). Якщо шаблон описаний усередині функції (всередині блоку {}) — це локальний шаблон, видимий лише з меж даної функції (даного блоку). Якщо описи шаблону вміщено поза блоками, такий шаблон видимо в усіх функціях нижче від точки опису шаблону до межі файла. Не можна описувати шаблон з ключовим словом extern.

Опис шаблону структури може проводитися поза функцією на рівні директив препроцесора. Зовнішній опис шаблону робить його доступним для всіх функцій, що йдуть за його визначенням. Приналежність структурної змінної до зовнішнього типу залежить від того, де визначена структурна змінна, а не де визначений шаблон.

13.5. МАСИВ СТРУКТУР І ЙОГО ОПИС У ТЕКСТІ ПРОГРАМИ

Масив, елементами якого є структури, називається *масивом структур*. Опис масиву структур відбувається за аналогією з описом окремої структури в два етапи, при цьому опис шаблону нічим не відрізняється, а при визначенні структурної змінної після її імені вказується розмірність у квадратних дужках, тобто

struct <ім'я шаблону> <ім'я структурної змінної>[розмірність];

Наприклад,

struct stt d[100];

визначений масив d, що складається зі 100 елементів типу stt. У цьому разі компілятор виділяє пам'ять для ста елементів d, довжина кожного з яких є сумою довжин елементів структури stt.

13.6. ВКЛАДЕНІ СТРУКТУРИ

У мові допускається використання вкладених структур, тобто структур, елементи яких у свою чергу є структурами. Елемент, що є структурою, називається вкладеною структурою. Наприклад,

```
struct name
    { char i[size1];        // ім'я
      char o[size2];        // по батькові
      char f[size3];};      // прізвище
struct stt {                // вкладена структура
        struct name fio; // шаблон вкладеної структури name
        char nz[size2];  // повинен бути вже відомий компілятору
        float sum;
       };
...
struct stt d;
```

Обмеження на вкладену структуру полягає в тому, що структура не може вкладуватися сама у собі. Тому наступний вираз не є коректним

```
struct stt {
   struct stt fio;
   char nz[size2];
   float sum; };
```


13.7. ЗВЕРНЕННЯ ДО ЕЛЕМЕНТІВ СТРУКТУРИ ТА МАСИВІВ СТРУКТУР

При зверненні до елементів структури необхідно зазначити в потрібному місці виразу чи оператора не лише ім'я елемента, а й ім'я структурної змінної, яка визначена раніше. Формат звернення є таким:

< ім'я структурної змінної >.< ім'я елемента>

Наприклад, для того, щоб звернутися до елемента *сума стипендії*, необхідно записати d.nz.

При звертанні до елементів масиву структур додатково до попереднього у форматі необхідно вказувати конкретний індекс структурної змінної, тобто

< ім'я структурної змінної >[індекс].< ім'я елемента>

Наприклад, d[5].nz.

Посилання на полі вкладеної структури формується з імені структурної змінної, імені структурного елемента, імені елемента вкладеної структури. Перелічені назви розділяються крапкою. Перелічені імена розділяються символом (.) операції крапка. Формат звернення до поля вкладеної структури є таким:

< ім'я структурної змінної >.<ім'я структурного елемента>. < ім'я елемента>

Наприклад, d.fio.i = "Олена"

Проілюструємо роботу з масивом структур на програмі, яка здійснює введення відомостей про стипендію студентів і їх виведення на екран.

```
// Введення даних про студентів
#include <stdio.h>
#define size 30
#define size1 7
#define kol 30
struct stt {   // оголошення шаблону
    char fio[size];        // ПІБ студента
    char nz[size1];        // N залікової книжки
    float sum;             // сума стипендії
};
void main( )
struct stt d[kol];         // оголошення масиву структур
int i; float sum_st;
for (i = 0; i<kol; i++)
 {printf ("\nВведіть прізвище: ");
  scanf ("%s", &d[i].fio);
  printf ("\n Введіть номер залікової книжки: ");
  scanf ("%s", &d[i].nz);
  printf ("\nВведіть суму стипендії: ");
  scanf ("%e",  &sum_st);
  d[i].sum = sum_st;
 }
// Виведення  даних
for (i = 0; i<kol; i++)
 printf ("\n %29s %6s %4.2f", d[i]. fio, d[i].nz, d[i].sum);
}
```

13.8. ВКАЗІВНИК НА СТРУКТУРНУ ЗМІННУ

Звертання до елементів структури можна організовувати за іменем елемента і за допомогою вказівника. При цьому всередині функції повинна бути описана змінна типу «вказівник на структуру» за таким форматом:

struct <ім'я шаблону>*<ім'я вказівника>;

Ця змінна перед зверненням до структури або масиву структур повинна бути встановлена на потрібну структуру:

<ім'я вказівника> = &ім'я структурної змінної [значення індексу];

Тільки після цього є можливим звернення до елемента:
ім'я вказівника –>ім'я елемента

Приклад. Скористаємося умовою попередньої задачі.

```
#include<stdio.h>
#define size 30
#define size1 7
#define kol 5
struct stt {
     char fio[size];
     char nz[size1];
     float sum;
     };
main( )
{struct stt d[kol];
struct stt *pa;          // оголошення вказівника pa на структуру
                         // з шаблоном stt
int i; float sum_st;
for (i = 0;i<kol;i++)
{ printf ("\nВведіть прізвище: ");
  scanf ("%s",  &d[i].fio);
  printf ("\nВведіть номер залікової книжки: ");
  scanf (" %s",  &d[i].nz);
  printf ("\nВведіть суму стипендії: ");
  scanf ("%e",  &sum_st);
  d[i].sum=sum_st;
}
// вказівнику pa присвоюється початкова адреса масиву структур d
pa = &d[0];
for (i = 0; i<kol; i++)
 {printf("\n%29s  %6s   %4.2f", pa–>fio,  pa–>nz,  pa–>sum);
  pa++;}
}
```

При використанні вказівників на структуру звернення до елемента структури проводиться таким чином: pa–>sum. Цей вираз означає, що ми звертаємося до третього елемента структури з масиву d. Щоб отримати адресу другої структури з масиву структур d, треба вказівник pa, який містить адресу першої структури з масиву d, збільшити на одиницю. Для цього використовується операція інкремента pa++;.

13.9. ВИКОРИСТАННЯ СТРУКТУР У ФУНКЦІЯХ

Вказівник часто використовується для передачі даних, об'єднаних у структури між функціями. Це пояснюється тим, що в списках аргументів і параметрів не дозволяється використовувати ні імені типу структури, ні структурної змінної, а лише імена окремих елементів структури. Жодних обмежень не накладається на передачу адрес окремих елементів структури.

Якщо продовжити попередній приклад, реалізовуючи додатково до введення даних по групі студентів функцію підрахунку загальної стипендії, алгоритм якої реалізований окремо в функції sums, то звернення до цієї функції з викликаючої функції матиме ось який вигляд:

```
for (i = 0; i<kol; i++)
   sums (d[i].sum);
```

Передача адреси окремого елемента має вигляд:

```
sums (&d[i].sum);
```

Для пересилання у функцію інформації структури в цілому необхідно організувати передачу адреси структурної змінної даної структури. З цією метою в списку аргументів викличної функції формується адреса структури:

ім'я функції (&ім'я структурної змінної);

У заголовку викликуваної функції в списку параметрів має бути представлений параметр, що описується як вказівник на ім'я типу структури:

<тип функції> <ім'я функції> (<вказівник>)

<ім'я вказівника> <ім'я типу структури> *<ім'я вказівника>;

Тільки після такої організації передачі структури можливо звертатися до її елементів за допомогою вказівника:

<ім'я вказівника> –> <ім'я елемента структури в шаблоні>

Для пересилання в функцію масиву структур необхідно пам'ятати, що ім'я будь-якого масиву є синонімом його адреси і тому ім'я масиву структур можна передавати у функцію, безпосередньо вказуючи його в списку параметрів.

У списку параметрів функції, яку викликають, дається вказівник, який за аналогією з одиничною структурою налаштовується на ім'я типу структури. Звернення до елементів відбувається як і при одиничній структурі, але попередньо повинна бути передбачена переадресація вказівника. Наприклад, у функції sums( ) робиться підрахунок суми стипендій по групі.

```
#include<stdio.h>
#define size 30
```



```
#define size1 7
#define kol 5
struct stt {
      char fio[size];
      char nz[size1];
      float sum;
      };
main( )
{struct stt d[kol];
 struct stt *pa;
 int i; float sum_st;
 void sums (struct stt *pa);
 for (i = 0;i<kol;i++)
{ printf ("\nВведіть прізвище: ");
  scanf (" %s",  &d[i].fio);
  printf ("\nВведіть номер залікової книжки: ");
  scanf (" %s",  &d[i].nz);
  printf ("\nВведіть суму стипендії: ");
  scanf ( "%e",  &sum_st);
  d[i].sum=sum_st;
}
pa = &d[0];
sums (pa);
}
void sums (struct stt *pr)
{float s; int i;
 for(i = 0, s = 0;i<kol;i++, pr++)
      s+ = pr–>sum;
 printf ("\nСумма стипендії по групі: %4.2f", s);
}
```

Адреси структурної змінної можна отримати, використовуючи операцію &, тоді pa = &d[0]; вміщує адресу масиву структур d у вказівник pr. Якщо передати цю адресу в функції, то вона отримає доступ до всіх елементів структури. Для доступу до членів структури за допомогою вказівника на структуру використовується операція –>, для доступу до елемента sum з допомогою pr записано: pr–>sum.

Розділ 14

ТЕХНІКА ОБРОБКИ ФАЙЛІВ ДАНИХ З ВИКОРИСТАННЯМ ЗАСОБІВ ПОТОКООРІЄНТОВАНОГО ВВЕДЕННЯ-ВИВЕДЕННЯ

14.1. ЗАГАЛЬНА ХАРАКТЕРИСТИКА ПЕРЕДАВАННЯ ФАЙЛІВ ПОТОКУ. ПОТОКИ І ФАЙЛИ

На відміну від інших мов програмування C++ не має команд введення-виведення, а реалізація введення-виведення в програмах здійснюється на базі функцій. Borland C++ має чотири системи введення-виведення:

1. Потокове введення-виведення (систему введення-виведення ANSI C);
2. Низькорівневе введення-виведення (систему введення-виведення UNIX);
3. Консольне введення-виведення;
4. Об'єктно-орієнтовану систему введення-виведення C++ (бібліотеку класів введення-виведення C++).

Система введення-виведення C і C++ грунтується на концепції потоків і файлів. *Потоком* називається абстрактне поняття, що ставиться до будь-якого перенесення даних від джерела (або постачальника даних) до приймача (або споживача даних). Потік введення-виведення — це логічний пристрій, який видає і приймає інформацію користувача. Потік визначає послідовність байтів (символів) і з погляду програми не залежить від тих конкретних пристроїв (файл на диску, принтер, клавіатура, дисплей і т. ін.), з якими ведеться обмін даними. Потік зв'язується з конкретним файлом за допомогою операції відкриття, зв'язок потоку з файлом знищується за допомогою операції закриття.

Файл — це іменована ділянка пам'яті, яка може зберігати дані, програму чи будь-яку іншу інформацію. Засоби операційної системи для доступу до файлів призначені для виконання таких операцій:

1) створення файлів;
2) знищення файлів;
3) пошуку файла на диску;
4) читання-записування інформації з файла (у файл);

5) захист файлів від несанкціованого доступу;
6) відкриття файлів;
7) закриття файлів.

З погляду програміста файл у мові C++ розглядається як послідовність символів або байтів, що має початок і кінець. У файлі визначений вказівник записування-читання — потокова позиція послідовності байтів, до яких здійснюється доступ. При кожному читанні або записуванні вказівник автоматично переміщується вперед на перенесену кількість байтів, якщо при цьому кінець файла не досягнутий. Досягнення кінця файла здійснюється операційною системою, яка повідомляє про це в програму передачею спеціальної умови EOF (умова EOF — End-of-File). Умова EOF реєструється операційною системою в той момент, коли вказівник досягає значення, рівного розміру файла. Система програмування мовою C++ для читання і записування інформації з файлів використовує свої бібліотечні функції, деякі з них виконують читання так званих текстових файлів або текстових потоків.

Текстовий файл містить інформацію у вигляді символів коду ASCII. У ньому можна зберігати текстову, цифрову інформацію, початкові тексти програм, файли пакетної обробки. Кожен символ має зовнішнє і внутрішнє (двійковий код) представлення і відповідний йому внутрішній двійковий код таких видів:

1) коди літер, розділових знаків, цифр і т. д.;
2) коди символів форматування. Наприклад, «повернення каретки», «переведення рядка» і т. д. Ці символи найчастіше зустрічаються для керування, передусім пристроєм друку, як логічні мітки кінця рядка в файлі;
3) коди символів зв'язку — такі, як «початок тексту», «кінець тексту», «кінець передачі» і т. ін., які використовуються при передачі інформації каналами зв'язку.

Довжина рядка текстового файла нічим не обмежена. Рядок продовжується до тих пір, поки не зустрінуться символи «повернення каретки» і «переведення рядка». Стандартні програмні засоби, наприклад, текстові процесори, можуть обмежувати довжину рядка, наприклад, до 80 символів (бітів). У цьому режимі комбінація символів «повернення каретки» і «переведення рядка» транслюється в єдиний символ «переведення рядка» при введенні. При виведенні символ введення рядка транслюється назад у комбінацію символів «повернення каретки» і «переведення рядка». У ряді випадків при введенні-виведенні небажано здійснювати такі перетворення. У двійковому режимі така трансляція комбінації подавляється.

Двійковий файл — послідовність байтів, що мають однозначну відповідність з байтами в зовнішньому пристрої.

Функції передавання даних потоку здійснюють обробку даних як безперервного потоку символів. Вони дозволяють записувати дані з будь якого зовнішнього запам'ятовувального пристрою (ЗЗП) в ОЗП і назад. Потокові функції виконують додаткову буферизацію на рівні операційної системи і на рівні бібліотечних функцій. Потокові функції введення-виведення забезпечують буферизовану обробку даних. Буфер — це спеціальна область ОЗП, яка виділяється для обслуговування процесу передавання даних. Використовування буфера забезпечує передачу даних у два етапи: на першому — дані з робочих полів ОЗП за командами виведення програми переміщуються до відповідного буфера виведення; на іншому етапі, коли буфер повністю заповнений, дані переміщуються на зовнішній ЗП. При введенні даних з ЗЗП в ОЗУ процес проходить аналогічно. При використанні буфера досягається вища швидкість виконання програми за рахунок скорочення кількості реальних звертань до ЗЗП. У той же час у процесі виконання програми потрібно стежити, щоб дані не залишилися в буфері на проміжній стадії обробки. Це може бути в процесі її введення-виведення, коли буфер залишається неповністю заповненим, або в процесі аварійної зупинки програми, коли буфер не оновлюється і дані можуть бути втрачені. При введенні даних із зовнішнього ЗП в ОЗП процес відбувається аналогічно.

Функції введення-виведення потоку забезпечують форматоване введення-виведення з проведенням редагування. Для функцій припускають посимвольне введення-виведення. Для роботи з функціями введення-виведення потоку до програми потрібно під'єднати ***<stdio.h>***.

14.2. ВИДАЛЕННЯ І ПЕРЕЙМЕНУВАННЯ ФАЙЛІВ

У мові C++ є ціла група стандартних функції, яка дозволяє керувати файлами. Одним з аргументів цих функцій обов'язково використовується ім'я файла, для якого необхідно виконати певні дії.

Ось перелік основних функцій вказаної групи:

• mktemp — створення унікального імені файла, генерує унікальне ім'я файла. Прототип функції міститься в файлі **<dir.h >** і має такий вигляд:

char *mktemp (char *template);

де char *template — вказівник на рядок, який повинен завершуватися.

Функція утворить унікальне ім'я файла, що не збігається з жодним з імен у поточному директорії, за допомогою модифікації заданого імені *template*. Рядок імені представляється у вигляді baseXXXXXX, де base — набір, що містить не більше шести будь-яких символів, які обов'язково включаються в ім'я, що генерується. Символи 'X' означають позиції, які будуть замінені символами унікального імені, причому третій з них завжди символ '.'. Функція *mktemp( )* починає генерацію імен з комбінації AA.AAA. Функція повертає вказівник на ім'я файла, у разі помилки повертає NULL. Функція не створює файла. Наприклад,

```
#include<dir.h>
#include<stdio.h>
 int main(void)
 {
  /* fname визначає маску для імені файла */
  char *fname = "TXXXXXX", *ptr;
  ptr = mktemp (fname);
  printf ("%s\n",ptr); // Виведення імені TAA.AAA
  return 0;
 }
```

• remove — видалення файла. Прототип функції знаходиться у файлі **<stdio.h>**.

int remove(const char *filename);

де const char *filename — вказівник на рядок, який містить ім'я файла, що видаляється, й може включати його повний маршрут.

Перед використанням функції *remove* файл, що вилучається, повинен бути закритим. Якщо файл вилучений — повертає 0, якщо помилка повертає – 1 і встановлює змінну *errno*, що дорівнює EACCES (доступ заборонений) та ENOENT (файл або каталог не знайдений). Функція реалізована як макрос, що викликає функцію unlink( ).

• rename — перейменування файла. Прототип функції знаходиться у файлі **<stdio.h>**.

int rename (const char *oldname, const char *newname);

де const char *oldname — вказівник на рядок, що містить ім'я файла;

const char *newname — вказівник на рядок, що містить нове ім'я.

Рядок, що задає ім'я, може містити ім'я дисковода й каталога. Функція *rename* може бути використана для переміщення файла з

одного каталога в інший, задавши інше ім'я шляху в *newname*. При перейменуванні файлів вони заздалегідь закриваються. Вказівка диску повинна бути одна й та сама. Функція *rename* повертає 0 при перейменуванні файла, – 1 у разі помилки і встановлює змінну *errno*, дорівнює ENOENT (файл або каталог не знайдений), EACCESS (доступ заборонений) та ENOTSAM (різні дисководи).

14.3. СТАНДАРТНІ ПОТОКИ ВВЕДЕННЯ-ВИВЕДЕННЯ

На початку виконання будь-якої програми система автоматично відкриває п'ять зумовлених потоків (які інакше називаються стандартними файлами):

stdin — стандартне введення;
stdout — стандартне виведення;
stderr — стандартне виведення повідомлень про помилки;
stdaux — стандартний послідовний порт;
stdprn — стандартний пристрій друку.

Дані потоки не треба явно відкривати й закривати. За замовчуванням *stdin, stdout, stderr* орієнтовані на дисплей, *stdprn* — на принтер, *stdaux* за замовчуванням не задається, а залежить від конкретної конфігурації системи, звичайно ці потоки зв'язують з послідовним портом і принтером. Операційні системи Windows і DOS допускають перенаправлення введення-виведення, це означає, що інформація, яка надходить звичайно на один пристрій, перенаправляється на інший пристрій операційної системи.

14.4. КЕРУВАННЯ БУФЕРИЗАЦІЄЮ

Буферизація потоків *stdin, stdout, stdprn* відбувається за замовчуванням завжди, а потоків *stdaux, srderr* лише в тому разі, коли вони зустрілись у відповідних функціях введення-виведення. Потоки *stdin, stdout, stderr* за замовчуванням відкриваються в текстовому режимі, а *stdaux* і *stdprn* — у двійковому. Для зміни режиму можна використовувати функцію *setmode( )*. Прототип функції є таким:

int setmode (int handle, int mode);

int handle — поточний номер потоку (дескриптор) встановлюється за допомогою функції *fileno( )*.

int fileno (FILE * stream);

де FILE *stream — відкритий файловий потік;

Для завдання режиму використовуються символьні константи: O_TEXT, O_BINARY. Якщо трапилася помилка — функція повертає значення – 1.

Функції потокового введення-виведення використовують вбудовану буферизацію для піднесення ефективності обміну з зовнішніми пристроями. Розмір внутрішнього буфера бібліотечних функцій Borland C++ за замовчуванням встановлюється 512 Кб для регулярних файлів (регулярний файл — це звичайний файл на диску), 128 Кб для стандартних файлів *stdin, stdout, stderr*. Використовування механізму буферизації пов'язане з можливістю втрати інформації в буфері в разі несподіваного (неочікуваного) або помилкового закінчення програми. У стандартній бібліотеці є дві функції, що дозволяють або повністю виключити буферизацію, або задати необхідний розмір буфера, або використовувати як буфер ділянку пам'яті, що надається прикладною програмою. Для створення буферів потоків *stdout, stdaux, stderr* можна використати функції керування буферизацією *setvbuf( ) і setbuf( )*.

int setvbuf (FILE *stream, char *buf, int type, size_t len);

де FILE *stream — відкритий файловий потік;

char *buf — вказівник на буфер, це звичайно масив елементів типу char;

int type — тип буферизації;

size_t len — розмір нового буфера в *байтах*, але ні більше 32767. Тип size_t визначений в файлі **<stdio.h>** як беззнакове ціле.

Функція *setvbuf( )* здійснює заміну буфера введення-виведення файла. Дана функція повертає значення 0, якщо функція спрацювала правильно, ненульове значення, якщо сталася помилка. Аргумент *type* задає встановлений спосіб буферизації і задається такими константами:

_IONBF — буферизація виключається, тоді інші елементи ігноруються;

_IOFBF — повна буферизація, тобто передавання даних із буфера у файл відбувається після повного заповнення буфера;

_IOLBF — порядкова буферизація. Після кожного запису символа «новий рядок» відбувається для буфера введення — очищення, а для буфера виведення — перенесення вмісту у файл.

void setbuf (FILE *stream, char *buf);

де FILE *stream — вказівник відкритого потоку;

char *buf — вказівник на новий буфер, який буде використовуватися для введення-виведення. Розмір буфера визначений у

<stdio.h> константою BUFSIZ що дорівнює 512 байтам. Щоб відмінити буферизацію файла, потрібно зробити розмір буфера нульовим.

Функція призначена для керування буферизацією потоку. Не має значення, що повертається. Якщо значення аргументу *buf* дорівнює NULL (нульовий вказівник), то буферизація відміняється. Дана функція використовується в тому разі, коли необхідно відключити буферизацію повністю або як область для буфера використовувати область пам'яті 512 Кб.

Якщо буфер створений системою, то він недосяжний для користувача, оскільки не має імені. Якщо є необхідність звільнити у програмі буфер до його повного заповнення, то використовують функції *fflush( )* і *flushall( )*.

int fflush (FILE *stream);

де (FILE *stream) — відкритий файловий потік.

Функція *fflush( )* звільняє буфер потоку на відповідному пристрої незалежно від його заповненості. Повертає значення 0, якщо буфер оновлений, а якщо під час виконання функції сталися помилки, то повертає значення EOF.

int flushall (void);

Функція звільняє всі буфери відкритих потоків. Повертає значення кількості відкритих потоків у разі успішного завершення, значення EOF, якщо виявлені якісь помилки. Закриття файлів також приводить до очищення буферів.

14.5. ВІДКРИТТЯ І ЗАКРИТТЯ ФАЙЛА

Якщо користувач у програмі крім стандартних файлів повинен здійснити опрацювання власних файлів даних, то необхідно проводити їх відкриття та закриття за допомогою стандартних функцій: *fopen( ), freopen( ), fclose( ), fcloseall( )*. Функція відкриття повертає значення вказівника типу FILE, встановлений на файл (потік). Тому в описовій частині програми повинен бути присутнім опис вказівника на файл:

FILE *<ім'я вказівника на файл>;

Вказівник на файл — це вказівник на інформацію, що визначає різні параметри файла, включаючи його ім'я, вказівник поточної позиції в файлі, вказівник на буфер і його розмір та ін. Вказівник на файл ідентифікує конкретний дисковий файл і використовується потоком для виконання операції введення-виведення. Вказівник на файл — це змінна-вказівник типу FILE.

У потрібному місці програми записується функція відкриття файла:

<ім'я вказівника на файл > = fopen (const char *filename, const char *mode);

де const char *filename — вказівник на рядок імені файла, який може містити інформацію про дисководи та каталоги;

const char *mode — вказівник на рядок, який задає режим доступу до файла й може набувати таких значень:

"r" — відкрити файл тільки для читання, при цьому файл уже повинен існувати;

"w" — відкрити порожній файл для записування, якщо цей файл раніше існував, то його вміст віддаляється;

"a" — відкрити файл для додавання записів у кінець, якщо файл не існує, то він створюється;

"r+" — відкрити файл одночасно для читання та записування, файл повинен існувати;

"w+" — відкрити файл для читання та записування, якщо файл раніше існував, то його вміст знищується;

"a+" — відкрити файл для читання та додавання записів у кінець, якщо файл не існує, то він створюється.

Додатково до будь-якого з цих режимів можна додати символ t — якщо файл потрібно відкрити в текстовому режимі, символ b — у двійковому режимі.

У разі успішної роботи дана функція повертає вказівник типу FILE, який ідентифікує відкритий файл і використовується файловими функціями, а в разі помилки функція повертає NULL-вказівник.

При завершенні роботи з файлом повинна бути записана функція закриття файла:

int fclose (FILE *stream);

де stream — це вказівник на файл, повернений *fopen( )*.

Функція *flose( )* повертає значення 0, якщо потік закритий успішно. У разі помилки значення, що повертається, дорівнює значенню EOF, тобто повертається ознака кінця файла, визначена в файлі **<stdio.h>** як –1.

Якщо потрібно закрити всі відкриті файли, крім стандартних файлів введення-виведення, використовують функцію *fcloseall( )*.

int fcloseall (void);

Функція повертає загальне число закритих потоків, якщо виявлені які-небудь помилки при закритті файлів — EOF.

Якщо потрібно відкрити файл з іншими правами доступу, то слід використовувати функцію:

FILE *freopen (const char *filename, const char *mode, FILE *stream);

де const char *filename — вказівник на рядок нового імені файла, на який перевизначається потік;

const char *mode — режим обробки для нового файла, що відкривається, визначається за аналогією з *fopen( )*;

FILE *stream — вказівник на відкритий потік, що закривається.

Функція закриває поточний відкритий файловий потік і зв'язує цей потік з новим файлом. У разі успіху функція повертає вказівник на знову відкритий потік. Якщо виникла помилка, колишній попередній файл закривається, а функція повертає значення NULL. Повторне відкриття того самого файла змінює права доступу, очищає внутрішні буфери і позиціонує вказівник поточної позиції або на початок файла ("r", "r+", "w", "w+"), або на його кінець ("a", "a+"). Якщо файли різні, перевідкриття приводить до переадресації файлового введення-виведення. Частіше за все ця функція використовується для перепризначення стандартних потоків *stdin, stdout, stderr* на інші файли. Наприклад,

freopen("out.txt", "w", stdout);

виконує переадресацію стандартного виведення в файл out.txt.

14.6. СКЛАД ФУНКЦІЙ ВВЕДЕННЯ-ВИВЕДЕННЯ ПОТОКОМ

Після успішного відкриття файла можна виконувати файлове введення-виведення. Функції потокового введення-виведення можна поділити на чотири групи:

1) функції посимвольного введення-виведення;
2) функції порядкового введення-виведення;
3) функції блокового введення-виведення;
4) функції форматованого введення-виведення.

Функції посимвольного введення. У разі успіху функції повертають значення символу. Якщо вказівник поточної позиції знаходиться за його кінцем або виникли якісь помилки, повертає EOF.

int fgetc (FILE *stream);

Функція *fgetc( )* читає один символ з потоку *stream* з поточної позиції і переміщує вказівник поточної позиції на наступний символ.

int fgetchar (void);

Функція *fgetchar* здійснює читання символу з стандартного файла *stdin (з* клавіатури), еквівалентна fgetc (stdin);.

int getc (FILE *stream);

Макрос *getc( )* повертає наступний за поточною позицією символ із заданого вхідного потоку *stream.*

int getchar (void)

Макрос *getchar( )* повертає символ з файла ***<stdio.h>***, еквівалентний getc (stdin);.

Функції порядкового введення. У разі успіху функції повертають вказівник на прочитаний рядок символів. У разі помилки або досягнення умови кінця файла вертає NULL.

char *fgets (char *s, int n, FILE *stream);

Функція *fgets* послідовно читає з файлового потоку FILE *stream символи аж до символа нового рядка, включаючи його, або максимум n–1 символів. Прочитані символи вміщуються, починаючи з комірки, що адресується аргументом char *s. До кінця рядка приєднується символ нового рядка, якщо він прочитаний, а також нульовий символ.

char gets (char *s);

Читає зі стандартного вхідного потоку символи, включаючи пробіли й табуляції, поки не зустрінеться символ нового рядка, який замінюється нульовим символом. Послідовність прочитаних символів вміщується в ділянку пам'яті, на яку вказує *s*.

Функції поблочного введення.

size_t fread (void *buf, size_t size, size_t n, FILE *stream);

де void *buf — вказівник на буфер, у який функція записує байти, прочитані з потоку;

size_t size — розмір у байтах одного елемента, що читається з файлового потоку;

size_t n — кількість елементів, що читаються;

FILE *stream — вказівник на відкритий файл;

Функція зчитує *n* елементів розміром *size* байтів кожен, в ділянку пам'яті, визначену вказівником *buf* з відкритого файла, на який вказує *stream*. У разі успішного виконання повертає число прочитаних елементів, а не байтів. Число прочитаних байтів дорівнює результату функції, помноженому на число байтів в елементі. У разі помилки чи досягнення кінця файла повертає EOF.

Функції форматованого введення. Повертає кількість аргументів, яким справді були присвоєні значення, повертає значення 0, якщо немає елементів, які були б присвоєні; EOF, якщо досягнуто кінця файла.

int fscanf (FILE *stream, const char *format [,address,...]);

де FILE *stream — вказівник на відкритий файл;

const char *format — рядок форматування, який задається аналогічно як у функції *scanf*;

address — список адресних вказівників, по одному для кожного зі специфікацією формату.

Функція *fscanf( )* читає дані з вказаного вхідного потоку *stream*, виконує форматування й отримані значення записує в змінні, адреси яких задаються аргументами *address*. Функція працює так само, як і функція *scanf( )*, за винятком того, що вона прочитує інформацію з потоку, вказаного аргументом *stream*, а не з *stdin*.

int sscanf (char *buffer, const char *format [, address,...]);

Функція *sscanf( )* аналогічна функції *scanf( )*, але дані прочитуються з масиву, вказаного аргументом *buffer*, а не зі стандартного потоку ***<stdio.h>***.

Функції посимвольного виведення. У разі успішного виконання повертає значення записаного символа. У разі помилки повертає EOF, оскільки для двійкових файлів значення EOF може бути просто одним з елементів файла; для визначення того, чи дійсно виникла помилка, необхідно використати функцію *ferror( )*. Аргументи символьного типу при виклику функції приводяться до цілого, звичайно як аргументи використовуються символьні змінні. Якщо використовується ціле число, то старший байт відкидається.

int fputc (int c, FILE *stream);

Вміщує символ, заданий *c*, у відкритий файл, на який вказує *stream*.

int putc (int c, FILE *stream);

Макрос *putc( )* записує символ, заданий *c*, в потік виведення, на який вказує *stream*.

int fputchar (int c);

Функція *fputchar( )* здійснює виведення символу в стандартний потік виведення (на екран).

int putchar (int c);

Макрос *putchar( )* записує символ, заданий *c*, в стандартний потік виведення *stdout*.

Функції порядкового виведення.

int fputs (const char *s, FILE *stream);

Функція *fputs( )* виводить символи з рядка, на який вказує *s*, у потік, на який вказує *stream*. Перенесення припиняється при досягненні символу кінця рядка, який у файл не переноситься.

int puts (const char *s);

Функція *puts( )* виводить символи з рядка, на який вказує *s*, у стандартний потік виведення stdout. Символ кінця рядка '\0' транслюється в символ нового рядка '\n'.

Функція поблочного виведення.

size_t fwrite(void *buf, size_t size, size_t n, FILE *stream);

де void *buf — вказівник на ділянку пам'яті, з якої функція записує байти в потік;

size_t size — розмір у байтах одного елемента, що записуються в файловий потік;

size_t n — кількість елементів даних, що записуються, кожний розміром *size* байтів;

FILE *stream — вказівник на відкритий файл у двійковому режимі;

Здійснює виведення неформатованих даних у потік, записування *n* елементів розміром *size* байтів кожний з області пам'яті, визначеної аргументом *buf* у відкритий файл, вказаний *stream*. Функція повертає кількість дійсно записаних елементів. Число записаних байтів дорівнює результату функції, помноженому на значення аргументу *size*. У разі помилки або досягнення кінця файла повертає EOF.

Функції форматованого виведення.

int fprintf (FILE *stream, const char *format [, argument,...]);

де FILE *stream — вказівник на відкритий файл;

const char *format — рядок форматування, який задається аналогічно як у функції *printf*;

argument — список аргументів, по одному для кожного зі специфікації формату.

Здійснює виведення форматованих даних у потік або відкритий файл. Повертає кількість записаних у потік байтів, якщо виявлена помилка, повертає EOF. Рядок *format* і аргументи задаються аналогічно функції *printf( )*.

int sprintf (char buffer, const char *format [, argument,...]);

Функція *sprintf( )* ідентична *printf( )*, за винятком того, що виведення форматованих даних призначається символьним масивом, адреса якого задається аргументом *buf*. Повертає число вміщених у цей масив символів, не включаючи завершальний нуль.

14.7. ФУНКЦІЇ ПОЗИЦІОНУВАННЯ

Місце у файлі, куди заноситься інформація або звідки вона читається, визначається значенням вказівника поточної позиції записування-читання файла. Бібліотека C++ здійснює доступ до будь-якого байта файла. При виконанні послідовного доступу вказівник позиціонується автоматично при будь-якій операції читання або записування.

int fseek (FILE *stream, long offset, int fromwhere);

FILE *stream — вказівник на відкритий файловий потік;

long offset — це вираз у байтах зсуву (зміщення) від позиції, що визначається *fromwhere*. Щоб перемістити вказівник поточної позиції в зворотному напрямі, у напрямі до початку файла, необхідно встановити *offset* так, щоб він дорівнював від'ємному значенню.

int fromwhere — задає точку відліку для зсуву.

Функція переміщує вказівник поточної позиції в файлі на нове місце в файлі, яке обчислюється за зміщенням і вказівкою напряму відліку. Аргумент *fromwher* може приймати значення таких констант:

0 або SEEK_SET — зсув виконується від початку файла;

1 або SEEK_CUR — зсув виконується від поточної позиції вказівника;

2 або SEEK_END — зсув виконується від кінця файла.

Якщо зміщення більше нуля, вказівник поточної позиції файла переміщується в бік кінця файла, якщо зміщення менше нуля, то переміщення вказівника виконується на початок файла. У разі успішного виконання функція повертає значення 0, у разі помилки — EOF.

long int ftell (FILE *stream);

Повертає положення вказівника поточної позиції файла, пов'язаного з потоком *stream*. Позиція видається як зміщення відносно початку файла. Значення, що повертається, має тип long int. У разі помилки функція повертає – 1L.

void rewind (FILE *stream);

Переміщення вказівника поточної позиції в початок вказаного файла *stream*. Ця функція очищає прапорець помилки і прапорець індикації кінця файла.

int fgetpos (FILE *stream, fpos_t *pos);

Визначає поточне положення вказівника файла, який вміщується за адресою пам'яті, що задається вказівником *pos*. Тип fpos_t визначений у файлі **<stdio.h>** як typedef long. У разі успіху дана функція повертає нуль, у разі помилки – 1 встановлює *errno*, що дорівнює EBADF (неправильний номер файла) або EINVAL (неправильний аргумент).

int fsetpos (FILE *stream, const fpos_t *pos);

Встановлює вказівник поточної позиції в файлі у відповідності зі значенням **pos*. У разі успіху повертає нуль, у разі помилки — ненульове значення.

Дві останні функції *fgetpos( )* і *fsetpos( )* використовуються для збереження і наступного відновлення позиції вказівника.

Для отримання інформації про поточний стан відкритого файла використовують функції перевірки закінчення файла та обробки помилок.

14.8. ФУНКЦІЯ ПЕРЕВІРКИ ЗАКІНЧЕННЯ ФАЙЛА

Для отримання поточного стану відкритого файла використовується функція *feof*.

int feof (FILE *stream);

Функція визначає, чи досягнутий кінець даного потоку, на який вказує *stream*. Якщо досягнутий кінець файла, то операція читання повертатиме символ кінця файла (значення EOF), поки не буде викликана функція *rewind()*.

14.9. ФУНКЦІЇ ОБРОБКИ ПОМИЛОК

int ferror (FILE *stream);

Здійснює перевірку наявності помилок при читанні-записуванні для заданого потоку, на який вказує *stream*. Якщо встановлена ознака помилки потоку, то вона зберігає її до моменту закриття потоку або до виклику функцій *clearerr( )* і *rewind( )*. Повертає ненульове число, якщо для відкритого файла, на який вказує *stream*, виникла помилка, в іншому разі повертає нуль

int clearerr (FILE *stream);

Очищає ознаку помилки та ознаки кінця файла для вказаного потоку.

Розглянемо описані функції на прикладі. Скажімо, потрібно обчислити кількість працівників, що мають безперервний загальний стаж роботи до 5 років, від 5 до 10 років, від 10 до 20 і більше років. Необхідно створити файл даних на магнітному диску, який міститиме таку інформацію: код підприємства, код цеху, табельний номер працівника, стаж загальний, стаж безперервний, розряд. Вихідну інформацію слід представити ось у якій формі:

Код підприємства	Кількість працівників з безперервним стажем			
	до 5 років	від 5 до 10	від 10до 20	понад

```
// Створення файла даних
#include <stdio.h>
#include <conio.h>
// Оголошення структурного шаблона
struct sv {
      int vz;
      char kpr[6];
```

```
      char kc[3];
      char tab[6];
      char gr[3];
      int sto;
      int stn;
      char razr[2];};
void main( )
{ struct sv v;      // Оголошення структурної змінної
struct sv vi;
  FILE *fl;         // Оголошення вказівника на файл
  int priz = 0;
  clrscr( );
  fl = fopen ("sved1", "w+");
  printf ("\n Продовження — 1");
  do { v.vz = 0;
       printf("\n Введіть код підприємства ");
       scanf ("%s",&v.kpr);
       printf ("\n Введіть код цеху ");
       scanf ("%s",&v.kc);
       printf ("\n Введіть табельний номер ");
       scanf ("%s",&v.tab);
       printf ("\n Введіть рік народження ");
       scanf ("%s",&v.gr);
       printf ("\n Введіть загальний стаж");
       scanf ("%d",&v.sto);
       printf ("\n Введіть безперервний стаж");
       scanf ("%d",&v.stn);
       printf ("\n Введіть розряд ");
       scanf ("%s",&v.razr);
       fwrite (&v,sizeof(v),1,fl);
       printf ("\n Введіть ознаку продовження");
       scanf ("%d",&priz);
}
while (priz = = 1); // Цикл формування запису
clrscr ( );
rewind (fl);
do
{fread(&vi,sizeof(vi),1,fl);
 if(!feof(fl))
    {printf("\n%2d%6s%3s%6s%3s% 2d% 2d%2s",vi.vz,vi.kpr, vi.kc,
                        vi.tab,vi.gr,vi.sto,vi.stn,vi.razr);
getch();}
}
```



```
while (!feof(fl));
fclose (fl);
}
// Обробка файла
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>
 struct sv {
       int vz;
       char kpr[6];
       char kc[3];
       char tab[6];
       char gr[3];
       int sto;
       int stn;
       char razr[2];};
 struct itog {
       char kpr[6];
       int s1;
       int s2;
       int s3;
       int s4;
  };
main( )
{ struct sv v;
  struct itog i;
  FILE *fl;
clrscr( );
fl = fopen ("sved1","r");
// Виведення шапки вихідного документа.
printf ("\n———————————————————————————");
printf ("\n| код   |кол-во раб. с непр. стаж. |");
printf ("\n|предпр. |до 5| 5-10|10-20| свыше |");
printf ("\n———————————————————————————");
fread (&v,sizeof(v),1,fl); // Читання першого запису з файла.
// Обробка прочитаного запису
while (feof(fl) = = 0)
{ stpcpy (i.kpr,v.kpr);
  i.s1 = 0; i.s2 = 0; i.s3 = 0; i.s4 = 0;
  while (feof(fl) = = 0 && atoi(v.kpr) = = atoi(i.kpr))
  {
```

```
    if (v.stn<5)
       i.s1++;
    else if (v.stn> = 5 &&v.stn<10)
       i.s2++;
    else if (v.stn> = 10 &&v.stn<20)
       i.s3++;
    else
       i.s4++;
  fread (&v,sizeof(v),1,fl); // Читання чергового запису
  if (!feof(fl) && v.vz = = 1)
    while (!feof(fl) && v.vz = = 1)
       { fread (&v,sizeof(v),1,fl);
         if (feof(fl)) break; }
   if (feof(fl) = = 0 && atoi(v.kpr)! = atoi(i.kpr))
     { printf ("\n| %5s | %d | %d | %d | %d |",
                 i.kpr,i.s1,i.s2,i.s3,i.s4);
     }
if (feof(fl)! = 0)
  { printf("\n| %5s | %d | %d | %d | %d |",
            i.kpr,i.s1,i.s2,i.s3,i.s4);
    printf("\n-------------------------------");
  }
}}
getch();
fclose(fl);
}
```

Розділ 15

ТЕХНІКА ОБРОБКИ ФАЙЛІВ ДАНИХ З ВИКОРИСТАННЯМ ЗАСОБІВ НИЗЬКОРІВНЕВОГО ВВЕДЕННЯ-ВИВЕДЕННЯ

15.1. ЗАГАЛЬНА ХАРАКТЕРИСТИКА НИЗЬКОРІВНЕВОГО ПЕРЕДАВАННЯ ДАНИХ

Бібліотека мови C++ підтримує UNIX-подібну систему введення-виведення. Функції низькорівневого введення-виведення не виконують буферизацію і форматування даних, це означає, що введення-виведення відбувається на елементарнішому рівні, ніж потокозорієнтований, фактично на рівні операційної системи. Для потокового оброблюваного файла у програмі не допускається змішення функцій потокового та низькорівневого введення-виведення.

Якщо файл відкривається як низькорівневий, то йому відповідає *дескриптор (handle)* — ціле число (порядковий номер), що характеризує розміщення інформації про відкритий файл у внутрішніх таблицях системи. Значення елемента цієї таблиці є порядковим номером у масиві описів відкритих файлів. Дескриптор використовується в наступних операціях для посилання на файл. При виконанні будь-якої програми визначається п'ять стандартних потоків:

Потік	Значення дескриптора
stdin	0
stdout	1
stderr	2
stdaux	3
stdprn	4

Ці файли, як і в потоковому виді передавання, можуть використовуватися без попереднього відкриття. Прототипи функцій низькорівневого введення-виведення знаходяться у файлі **<io.h>**. Багато які з функцій низькорівневого введення-виведення у разі помилки заздалегідь встановлюють певну глобальну цілочислову змінну *errno* у відповідний код помилки.

15.2. ВІДКРИТТЯ ТА ЗАКРИТТЯ ФАЙЛА

Для роботи з файлами даних користувача їх потрібно відкривати за допомогою функцій *open, sopen, creat.* Файл може бути відкритий у текстовому або двійковому режимах для записування, читання і для записування та читання одночасно. Функція відкриття повертає дескриптор файла, який можна присвоїти цілій змінній і потім використати цю змінну для посилань на файл. Функція *open* відкриває файл і підготовляє його для читання або записування, прототип функції є таким:

int open (const char *filename, int access [, unsigned mode]);

де const char *filename — ім'я файла, який відкривається, в подвійних лапках;

int access — режим доступу до файла, який визначається однією або комбінацією кількох констант, визначених у файлі **<fcntl.h>**. Якщо задано більше за одну константу, то вони розділяються з допомогою логічної операції АБО (|). Склад констант є таким:

O_APPEND — перемістити вказівник файла в кінець файла перед кожним зверненням для записування;

O_CREAT — створити новий файл і відкрити його для записування, не дає результату, якщо файл уже існує;

O_EXCL — використовується разом з O_CREAT і повертає значення помилки, якщо файл з таким ім'ям уже існує;

O_RDONLY — файл відкрити лише для читання, ні O_RDWR, ні O_WRONLY не можуть бути задані;

O_RDWR — файл відкрити для читання та записування, ні O_RDONLY, ні O_WRONLY не можуть бути задані;

O_TRUNC — файл відкрити і привести до довжини 0. Файл повинен мати дозвіл на запис, його вміст знищується;

O_WRONLY — файл відкрити лише для записування, ні O_RDONLY, ні O_RDWR не можуть бути задані;

O_BINARY — файл відкривається в двійковому режимі;

O_TEXT — файл відкривається в текстовому режимі.

Аргумент *mode* використовується в тому разі, якщо аргумент *access* заданий константою O_CREAT. Якщо файл існує, значення аргументу *mode* ігнорується. Він визначає спосіб доступу до файла, коли відкривається файл. Константи для аргументу *mode* знаходяться у файлі **<sys\stat.h>**.

S_IREAD — доступ для читання;

S_IWRITE — доступ для записування;

S_IREAD | S_IWRITE — доступ для читання і записування.

Якщо доступ не заданий, файл відкритий лише для читання.

Функція *open* повертає дескриптор на відкритий файл. У разі помилки повертає –1, це означає, що файл не може бути відкритий і встановлює у зовнішній змінної *errno* одне з таких значень: ENOENT — файл або каталог не знайдені; EMFILE — відкрито дуже багато файлів; EACCESS — недозволений вид доступу, EINVACC — невірний код доступу.

int sopen (const char *filename, int access, int shflag, unsigned mode);

Відкриває файл *filename* і готує його до наступного окремого читання та записування. Функція *sopen* повертає дескриптор для відкритого файла, у разі помилки повертає – 1. Аргумент *filename*, аргумент *access*, аргумент *mode* задаються аналогічно функції open. Аргумент *shflag* є константним виразом, який складається з констант, визначених у файлі **<share.h>**.

SH_DENYRW — заборонити доступ до файла для читання та записування;

SH_DENYWR — заборонити доступ до файла для записування;

SH_DENYRD — заборонити доступ до файла для читання;

SH_DENYNO — дозволити доступ до файла для читання та записування.

Функція *sopen* повертає дескриптор на відкритий файл. У разі помилки вертає -1 і встановлює у зовнішній змінної *errno* код помилки.

int creat (const char *filename, int amode);

Функція *creat* або створює новий файл, або відкриває та звужує до довжини 0 існуючий файл. Аргумент *filename* та аргумент *amode* задаються аналогічно функції *open*.

Якщо файл, що вказаний аргументом *filename*, не існує, створюється новий файл із заданим доступом і відкривається для записування. Якщо файл уже існує з доступом для записування, то функція звужує його до довжини 0, знищує попередні дані і відкриває його для записування. Якщо файл існує і доступний лише для записування, то функція *creat* вертає ознаку помилки і файл залишається незміненим. Права доступу до файла, що встановлюються аргументом *amode*, застосовуються тільки для новостворюваних файлів. Новий файл отримує вказані права доступу після того, як він буде закритий.

int access (const char *filename, int mode);

Функція *access* використовується для перевірки існування файла. Вона може також використовуватися для з'ясування, чи захищений файл від запису і чи є файлом, що виконується. Ім'я файла, що перевіряється, вказується за допомогою аргументу

filename. Аргумент *mode* визначає, що саме перевіряє *access*, і може набувати таких значень:

0 — файл існує;
1 — файл, що виконується;
2 — файл доступний для запису;
3 — файл доступний для читання;
4 — файл доступний для читання-записування.

Функція *access* повертає 0, якщо вид доступу, що перевіряється, дозволений; в іншому разі, повертає – 1. Глобальна змінна *errno* встановлюється в одне з таких значень: ENOENT — файл або каталог не знайдений, EACCESS — вказаний вид доступу не дозволений. Наприклад,

```
if (!access ("test.txt", 0))
   printf ("Файл існує");
else
   printf ("Файл не знайдений");
```

Закриття файла здійснюється функцією *close*.

int close (int handle);

Функція *close* закриває файл, пов'язаний з дескриптором *handle*. Функція повертає значення 0, якщо файл успішно закритий, значення –1 свідчить про помилку, при цьому змінної *errno* присвоюється EBADF — неправильний номер файла.

15.3. ЧИТАННЯ І ЗАПИСУВАННЯ ДАНИХ

Для читання і записування даних призначені дві функції: *read* і *write*.

int read (int handle, void *buf, unsigned len);

де int handle — дескриптор відкритого файла;

void *buf — вказівник на буфер, у який копіюються прочитані дані;

unsigned len — максимальна кількість прочитаних байтів.

Функція *read* виконує читання *len* байтів, з файла, пов'язаного з дескриптором *handle*, в ділянку пам'яті, адреса якої визначається значенням аргументу *buffer*. Операція читання починається з поточної позиції вказівника файла, пов'язаного із заданим файлом.

У разі успішного читання повертає кількість прочитаних байтів. Повернуте значення числа байтів може бути меншим від запитаного числа *len*, якщо файл є текстовим, або число байтів, що лишилися у файлі, менше, ніж len. При досягненні кінця

файла функція повертає 0, у разі помилки повертає – 1. Якщо сталася помилка, errno встановлюється в одне з таких значень: EACCESS — недозволений вид доступу, EBADF — неправильний номер файла.

Якщо файл був відкритий у текстовому режимі, повернуте значення для функції *read* може не відповідати числу дійсно прочитаних байтів. Це відбувається тому, що кожна комбінація символів <переведення каретки> <новий рядок> заміщується символом нового рядка, і тільки цей символ враховується у повернутому значенні.

int write (int handle, void *buf, unsigned len);

Функція *write* виконує запис *len* байтів у відкритий файл, пов'язаний з дескриптором *handle*. Дані, що записуються, знаходяться в оперативній пам'яті, починаючи з адреси, що задається вказівником *buffer*. Операція запису починається з поточної позиції вказівника файла. Після операції запису вказівник файла збільшується на число реально записаних у файл байтів. Якщо файл відкривається для додання записів, то виведення здійснюється в кінець файла.

Функція повертає число дійсно записаних байтів, обов'язково більше або рівне заданому значенню *len*. Для текстових файлів символ кінця рядка замінюється на послідовність <повернення каретки> — <переведення рядка>. Значення, що повертається –1 — вказує на помилку. Якщо виникла помилка, *errno* встановлюється в одне з наступних значень: EACCESS — недозволений вид доступу, EBADF — неправильний номер файла.

15.4. КЕРУВАННЯ ВКАЗІВНИКОМ ПОТОЧНОЇ ПОЗИЦІЇ

long lseek (int handle, long offset, int fromwhere);

int handle — дескриптор відкритого файла;

long offset — зміщення, визначене в байтах, якщо *offset* > 0, виконується зсув у бік кінця файла, *offset* < 0 — виконується зсув у бік початку файла;

int fromwhere — задає напрям відліку для зсуву і може приймати одне з таких значень:

SEEK_SET або 0 — зсув виконується від початку файла;

SEEK_CUR або 1 — зсув виконується від поточної позиції вказівника файла;

SEEK_END або 2 — зсув виконується від кінця файла.

Функція *lseek* у разі успішного виконання вертає зміщення в байтах нової позиції вказівника від початку файла, в разі помилки вертає –1L, *errno* встановлюється в одне з таких значень: EBADF — неправильний номер файла, EINVAL — неправильний аргумент.

long tell (int handle);

Функція *tell* використовується для отримання поточної позиції вказівника файла, пов'язаного з дескриптором *handle*. Позиція представляється як зміщення в байтах від початку файла. У разі помилки повертає значення – 1L, якщо задане неправильне значення аргументу *handle*, при цьому змінної *errno* присвоюється значення EBADF.

15.5. ПЕРЕВІРКА КІНЦЯ ФАЙЛА

Для аналізу умови кінця файла може використовуватися функція *eof*.

int eof (int handle);

Функція *eof* визначає, чи був досягнутий кінець файла, пов'язаний з дескриптором *handle*. Повертає 1, якщо досягнуто кінець файла; 0 — якщо кінець файла не досягнуто; – 1 — якщо виникла помилка, при цьому встановлює в *errno* значення EBADF.

Розглянуті функції проілюструємо прикладом створення файла функціями, низькорівневим введенням-виведенням.

```
// Створення файла
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <conio.h>
main()
{
int f, an;
unsigned char answ, answ1;
struct dat
{
 char pred[6], ceh[3], tab_n[5], fio[31], god[5], nats[3], sem[2],
 st_o[3], st_n[3], prof[5], razr[2], oklad[7];
}
db;
printf ("\n\n Що робити? \n 1—Переписати\n 2—Додати\n Введіть:");
scanf("%d", &an);
switch (an)
```



```
{
case 1: f = open ("data.hol", O_CREAT ¦ O_WRONLY);
 break;
case 2: f = open("data.hol", O_RDWR ¦ O_APPEND);
 break;
}
do
{
 do
 {
  printf ("\n\n Введіть код підприємства—");
  scanf ("%5s", db.pred);
  printf ("\n Введіть код цеху—");
  scanf ("%2s", db.ceh);
  printf ("\nВведіть номер робітника—");
  scanf ("%4s", db.tab_n);
  printf ("\nВведіть ПІБ робочого—");
  scanf ("%30s", db.fio);
  printf ("\nВведіть рік народження—");
  scanf ("%4s", db.god);
  printf ("\nВведіть код національності—");
  scanf ("%2s", db.nats);
  printf ("\nВведіть сімейний стан—");
  scanf ("%1s", db.sem);
  printf ("\nВведіть загальний стаж—");
  scanf ("%2s", db.st_o);
  printf ("\nВведіть безперервний стаж—");
  scanf ("%2s", db.st_n);
  printf ("\nВведіть код професії—");
  scanf ("%4s", db.prof);
  printf ("\nВведіть розряд—");
  scanf ("%1s", db.razr);
  printf ("\nВведіть оклад—");
  scanf ("%4s", db.oklad);
  printf ("Коректувати (y/n)?\n");
  scanf ("%s", &answ);
 }
 while (answ = = 'y' ¦¦ answ = = 'Y');
 an = write(f, &db, sizeof(db));
 printf ("ще вводити (y/n)?");
 scanf ("%s", &answ1);
 }
 while (answ1 = = 'y' ¦¦ answ1 = = 'Y');
 close (f);
}
```

Розділ 16

ФАЙЛОВЕ ВВЕДЕННЯ-ВИВЕДЕННЯ С++

16.1. ПОТОКИ ВВЕДЕННЯ-ВИВЕДЕННЯ С++

С++ підтримує всі функції введення-виведення С і, крім того, визначає свою власну об'єктно-орієнтовану систему введення-виведення. Система введення-виведення С++, як і С, оперує потоками.

Інтерфейс бібліотеки потоків *iostream* розбитий на кілька заголовних файлів:

<iostream.h> містить основні класи потокового введення-виведення;

<iomanip.h> визначає маніпулятори потокового введення-виведення, а також різні макроси для створення параметризованих маніпуляторів;

<fstream.h> містить оголошення потокових класів файлового введення-виведення;

<strstream.h> містить визначення потокових класів для байтових масивів пам'яті.

Бібліотека *iostream* містить багато класів для обробки широкого спектра операцій введення-виведення. При обробці файлів в С++ використовуються такі класи:

клас *ifstream*, що виконує операції введення з файла;

клас *ofstream*, що виконує операції виведення в файл;

клас *fstream*, призначений для операцій введення-виведення.

Клас *istream* і клас *ostream* є похідними класами прямого успадкування базового класу *ios*. Клас *iostream* є похідним класом множинного успадкування класів *istream* і *ostream*. Клас *ifstream* успадковує клас *istream*, клас *ofstream* успадковує клас *ostream*, клас *fstream* — клас *iostream*. Структура зв'язків успадкування класів потоків введення-виведення наведена на рис. 16.1.

Коли запускається програма С++, автоматично відкриваються чотири потоки: *cin* — стандартний потік введення, *cout* — стандартний потік виведення, *cerr* — небуферизований стандартний потік виведення про помилки, *clog* — буферизований стандартний потік виведення про помилки.

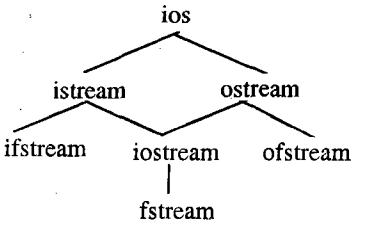


Рис. 16.1. Частина ієрархії класів потоків введення-виведення з ключовими класами обробки файлів

16.2. ВІДКРИТТЯ І ЗАКРИТТЯ ФАЙЛІВ. ВИЗНАЧЕННЯ КІНЦЯ ФАЙЛА

В С++ файл відкривається за допомогою його скріплення з потоком. Є три типи потоків: введення, виведення, введення-виведення. Перед тим як відкрити файл треба створити потік. Для створення потоку введення необхідно оголосити об'єкт типу *ifstream*. Для створення потоку — об'єкт типу *ofstream*. Потоки, які виконують одночасно введення і виведення, повинні оголошуватися як об'єкти типу *fstream*.

Після створення потоку, одним зі способів зв'язати його з файлом є функція *open( )*. Ця функція є членом кожного з трьох потокових класів. Вона має такий прототип:

void open (const char *filename, int mode);

де const char *filename є ім'ям файла і може включати вказівку шляху;

int mode — визначає режим відкриття файла, значення якого може бути одним з наступних, визначених у заголовному файлі ***<fstream.h>***:

ios::app — відкриття файла в режимі додання в кінець файла. Це значення може прийматися лише до файлів, що відкриваються для виведення;

ios::ate — викликає пошук кінця файла в момент відкриття файла, дані можуть бути записані в будь-яке місце файла;

ios::binary — викликає відкриття файла в двійковому режимі. За замовчуванням файли відкриваються в текстовому режимі;

ios::in — задає режим відкриття файла для введення;

ios::out — задає режим відкриття файла для виведення;

ios::trunc — видаляє вміст файла, що раніше існував з тією ж назвою, і усікає його до нульової довжини. При створенні потоку

виведення за допомогою ключового слова *ofstream* будь-який файл, що раніше існував з тим же ім'ям, автоматично усікається до нульової довжини.

Можна комбінувати два і більше з цих значень, використовуючи порозрядне АБО. Наприклад, оголошення

ofstream outFile;

створює об'єкт outFile класу *ofstream*.

outFile.open ("test.txt", ios::out);

Функція *open* класу *ofstream* відкриває файл і зв'язує його з існуючим об'єктом класу *ofstream*.

Класи *ifstream, ofstream* і *fstream* містять конструктори, що автоматично відкривають файл. Конструктори мають ті самі параметри і значення за замовчуванням, що й функція *open( )*. Тому найтиповішим способом відкриття файла є такий:

ofstream outFile ("test");

Відкрити файл для виведення, режим встановлюється за замовчуванням.

Якщо з якихось причин файл не може бути відкритий, значення асоційованого потоку дорівнюватиме NULL.

```
if (!outFile)
  {cerr << "Файл не може бути відкритий";
   exit(1);}
```

Для того, щоб закрити файл, необхідно використати функцію-член *close( )*. Наприклад,

outFile.close( );

Функція *close( )* не має параметрів і значення, що повертається.

Для визначення досягнення кінця файла можна використати функцію-член *eof( )*, яка має ось який прототип:

int eof();

Функція повертає істину, якщо був досягнутий кінець файла, в іншому разі функція повертає хибність.

16.3. ФОРМАТОВАНЕ ВВЕДЕННЯ-ВИВЕДЕННЯ

Після того, як файл відкритий, можна прочитувати і записувати в нього дані. Для текстових файлів можна використати оператори << і >>, як це робилося для консольного введення-виведення. Тільки необхідно замінити потік *cin* або *cout* на потік, з яким пов'язаний файл.

16.4. ВВЕДЕННЯ-ВИВЕДЕННЯ, ЩО НЕ ФОРМАТУЄТЬСЯ

Для прочитання та запису блоків даних використовуються функції *read( )* і *write( )*, які є членами потокових класів введення і виведення відповідно. Прототипи функцій такі:

istream &read (char *buf, streamsize num);

ostream &write (const char *buf, streamsize num);

Функція *read( )* прочитує *num* байтів із викликаючого потоку і передає їх у буфер, визначений вказівником *buf*. Тип *streamsize* визначений як ціле число. Якщо кінця файла досягнуть до того, як було прочитано *num* байтів, виконання функції припиняється, а буфер містить стільки символів, скільки було прочитано. Визначити, скільки було прочитано символів, можна, використовуючи функцію-член *gcout( )*, прототип якої є таким:

int gcout( );

Функція повертає кількість символів, прочитаних під час останньої операції введення.

Функція *write( )* записує у відповідний потік з буфера, який визначений вказівником *buf*, задане *num* число байтів.

16.5. ДОВІЛЬНИЙ ДОСТУП

Система введення-виведення C++ дозволяє здійснювати довільний доступ з використанням функцій *seekg( )* і *seekp( )*. Вона керує двома вказівниками, пов'язаними з файлом. Перший — це вказівник прочитання (*get point*), який визначає місце в файлі, куди буде вводитися інформація. Другий — це вказівник запису (*put point*), який задає наступне місце у файлі, куди буде виводитися інформація. Щоразу, коли здійснюються операції введення-виведення, відповідний вказівник автоматично пересувається. За допомогою функцій *seekg( ) і seekp( )* можна здійснювати доступ до файла в довільному місці.

istream &seekg (off_type offset, seek_dir origin);

Функція *seekg( )* переміщує вказівник прочитання на *offset* байтів від заданого *origin*, що набуває одне з таких значень:

ios::beg — пошук від початку файла;

ios::cur — пошук від поточної позиції файла;

ios::end — пошук з кінця файла.

ostream &seekp(off_type offset, seek_dir origin);

Функція *seekp( )* переміщує вказівник запису на *offset* байтів від заданого *origin*, що приймає одне з вищезгаданих значень.

Тип *off_type* — це цілий тип даних, визначений у класі *ios*, тип *seek_dir* — це перелік, визначений у класі *ios*.

Визначити поточну позицію кожного з двох вказівників можна за допомогою функцій:

pos_type tellg( );
post_type tellp( );

post_type — це цілий тип даних, визначений у класі *ios* і здатний зберігати якнайбільше значення вказівника.

16.6. КОНТРОЛЬ СТАНУ ВВЕДЕННЯ-ВИВЕДЕННЯ

У системі введення-виведення C++ підтримується інформація про стан після кожної операції введення-виведення. Поточний стан потоку введення-виведення, який зберігається в об'єкті типу *iostate*, є перелічувальним типом, визначеним у класі *ios*, і містить такі члени:

goodbit — помилки немає;
eofbit — досягнуть кінця файла;
failbit — має місце нефатальна помилка;
badbit — має місце фатальна помилка.

Одним з способів отримання інформації про стан введення-виведення є виклик функції *rdstate( )*. Прототип функції такий:

iostream rdstate( );

Функція повертає поточний стан прапорів помилки, при відсутності якої б то не було помилки повертається прапор goodbit, в іншому разі вона повертає прапор помилки, значення якого перераховані вище.

Іншим способом визначення того, чи мала місце помилка, є використання однієї чи кількох наступних функцій-членів класу *ios*:

boll bad( );
boll eof( );
boll fail( );
boll good( );

Функція bad( ) повертає істину, якщо встановлений прапор *badbit*. Функція *fail( )* повертає істину, якщо встановлений прапор *failbit*. Функція *good( )* повертає істину при відсутності помилок, в іншому разі повертає хибність.

Після появи помилки необхідно скинути цей стан перед тим, як продовжити виконання програми. Для цього використовується функція *clear( )*. Прототип функції є таким:

void clear (iostate = ios::goodbit);

де s — це прапор, що дорівнює *goodbit*, значення за замовчуванням, то скидаються прапори всіх помилок. В іншому разі змінній присвоюється значення тих прапорів, які необхідно скинути.

Приклад Потрібно створити файл даних на магнітному диску, який міститиме таку інформацію: код підприємства, код цеху, табельний номер працівника, рік народження, код національності, сімейний стан, стаж роботи загальний, стаж роботи безперервний, код професії, розряд, оклад. На основі запиту обчислити кількість та питому вагу вікових груп працівників (до 20 років, від 20 до 30, від 30 до 40 років та старших за 40 років) у розрізі кодів професій. Знайдені показники вивести в такій формі:

Код підприємства	Код цеху	Код професії	Вік, років							
			до 20		від 20 до 30		від 30 до 40		понад 40	
			кількість	відсоток	кількість	відсоток	кількість	відсоток	кількість	відсоток

```
// Створення файла
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <ctype.h>
struct work
{ char kpr [6]; char kc [3]; char tab [5];
  char namp [31]; char gr [5]; char naz [3];
  char fs [2]; char stz [3]; char stn [3];
  char prof [5]; char rozr [2]; float okl;
};
main ()
{ int r, b, a;
  work w;
  ofstream outwork ("test.dat", ios:: out);
  if (! outwork)
    { cerr <<"Файл не може бути відкритий."<<endl;
  exit (1);
}
```

```
do
{ cout <<"Введіть дані \n";
  a = 0;
do
  { b = 0;
do
{ cout <<"Введіть код підприємства:";
  r = 0;
  cin>> w.kpr;
  for (int p = 0;p<5;p++)
    if(!isdigit(w.kpr[p]))
      r = 1;
  if(r = = 1)
   cout<<"Помилка введення.\n";
}
while (r = = 1);
do
{ cout <<"Введіть код цеху:";
  r = 0;
  cin>> w. kc;
  for (int c = 0; c<2; c++)
   if (! isdigit (w.kc [c]))
      r = 1;
if (r = = 1)
cout <<"Помилка введення. \n";
}
while (r = = 1);
do
{ cout <<"Введіть табельний номер працівника:";
  r = 0;
  cin>>w.tab;
  for (int n = 0;n<4;n++)
   if (! isdigit(w.tab[n]))
      r = 1;
    if (r = = 1)
    cout<<"Помилка введення. \n";
}
while (r = = 1);
   cout <<"Введіть прізвище:";
   cin>>w.namp;
   do
   { cout <<"Введіть рік народження:";
     r = 0;
     cin >> w.gr;
     for (int y = 0;y<4;y++)
       if (! isdigit(w.gr[y]))
```



```
        r = 1;
       if (r = = 1)
       cout<<"Помилка введення. \n";
     } while (r = = 1);
do
 { cout <<"Введіть код національності:";
   r = 0;
   cin >>w.naz;
   for (int nac = 0;nac<2;nac++)
    if (! isdigit(w.naz[nac]))
      r = 1;
    if (r = = 1)
    cout<<"Помилка введення. \n";
}
while (r = = 1);
do
 { cout <<"Введіть сімейний стан:";
r = 0;
cin >>w.fs;
if (strcmp(w.fs,"Ж")! = 0 && strcmp(w.fs, "Х")! = 0)
r = 1;
if (r = = 1)
cout<<"Помилка введення. \n";
} while (r = = 1);
do
{ cout <<"Введіть стаж роботи загальний:";
  r = 0;
  cin >> w.stz;
  for (int t = 0;t<2;t++)
   if (!isdigit(w.stz[t]))
   r = 1;
    if (r = = 1)
    cout <<"Помилка введення. \n";
 } while (r = = 1);
do
{ cout <<"Введіть стаж роботи безперервний:";
  r = 0;
  cin >> w.stn;
 for (int d = 0;d<2;d++)
  if (!isdigit(w.stn[d]))
    r = 1
    if (r = = 1)
      cout<<"Помилка введення. \n";
 } while (r = = 1);
do
{ cout <<"Введіть код професії:";
```

```
  r = 0;
  cin >>w.prof;
  for (int pr = 0;pr<4;pr++)
    if (! isdigit(w.prof[pr]))
       r = 1;
    if (r = = 1)
       cout<<"Помилка введення \n";
} while (r = = 1);
do
{ cout <<"Введіть розряд:";
  r = 0;
  cin >>w.rozr;
  for ( int r = 0;r<1;r++) if(!isdigit(w.rozr[r]))
 r = 1;
 if (atoi(w.rozr)>6 || atoi(w.rozr)<1)
  r = 1;
 if (r = = 1) cout<<"Помилка введення. \n";
} while (r = = 1);
cout <<"Введіть оклад: ";
cin >>w.okl;
cout<<"Введіть 1, якщо під час введення зроблена помилка:";
cin >>b;
}
while (b = = 1);
outwork.write ((char *)&w,sizeof(w));
cout <<"Введіть 1 для продовження:";
cin >>a;
} while (a = = 1);
outwork.close();
return 0;
}
//Обробка файла
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <iomanip.h>
struct work
{ char kpr[6]; char kc[3]; char tab[5];
  char namp[31]; char gr[5]; char naz[3];
  char fs[2]; char stz[3]; char stn[3];
  cha prof[5]; char rozr[2]; float okl;
};
main ()
{ work w,wl;
```



```
  int year, p, t1 = 0, t2 = 0, t3 = 0, t4 = 0;
  float tlpr = 0, t2pr = 0, t3pr = 0, t4pr = 0;
  ifstream inwork ("test.dat",ios::in);
  if (!inwork)
  { cerr <<"Файл не може бути відкритим"<<endl;
  exit(l);
  } cout<<endl<<"Введіть код підприємства";
  cin >> wl.kpr;
  cout<<endl<<"\nВведіть код цеху";
  cin >>wl.kc;
  cout<<endl<<"\nВведіть код професії";
  cin>>wl.prof;
  cout<<endl<<"\nВведіть поточний рік";
  cin>>year;
  inwork.read ((char *)&w,sizeof(w));
  while (! inwork.eof())
  { p = year — atoi(w.gr);
  if(strcmp(wl.kpr,w.kpr) = = 0 && strcmp(wl.kc,w.kc) = = 0 &&
strcmp(wl.prof,w.prof) = = 0 && p<20)
     tl++;
  else
  if (strcmp(wl.kpr,w.kpr) = = 0 && strcmp(wl.kc,w.kc) = = 0 &&
strcmp(wl.prof,w.prof) = = 0 && p> = 20 && p<30)
     t2++;
  else
  if (strcmp(wl.kpr,w.kpr) = = 0 && strcmp(wl .kc,w.kc) = = 0 &&
strcmp(wl .prof,w.prof) = = 0 && p> = 30 && p<40)
     t3++;
  else
  if (strcmp(wl .kpr,w.kpr) = = 0 && strcmp(wl.kc,w.kc) = = 0 &&
strcmp(wl .prof,w.prof) = = 0 && p> = 40)
     t4++;
  inwork.read((char * )&w,sizeof(w));
  } inwork.close();
  tlpr = (float)(tl * 100/(tl+t2+t3+t4));
  t2pr = (float)(t2* 100/(tl +t2+t3+t4));
  t3pr = (float)(t3*100/(tl+t2+t3+t4));
  t4pr = (float)(t4* 100/(tl +t2+t3+t4));
  cout<<endl;
  cout<<"                                                                    "
      <<"                    "
      <<"          "
  <<endl;
  cout<<" |"<<setw(12)<<" "<<"|"<<setw(4) <<" "<<"|"<<setw(8)
  <<" "<<"|"<<setw(47)<<"BIK,POKIB        "<<"|"<<endl;
```

```
   cout<<"|"<<setw(12)<<"КОД  "<<"|"<<setw(4)<<"КОД          "
       <<"|"<<setw(8)<<"КОД "
<<"|"<<"_____|"
<<endl;
   cout<<" |"
<<setw(12)<<"ПІДПРИЄМСТВА"<<"|"<<setw(4)<<"ЦЕХУ"<<"|"
<<setw(8) <<"ПРОФЕСІЇ "<<"|"<<setw(11)<<"ДО 20"
   <<"|"<<setw(11) <<"ВІД 20"<<"|"<<setw(11)
   <<"ВІД 30"<<"|"<<setw(11)<<"ПОНАД 40"<<"|"<<endl;
   cout<<" |"<<setw(12)<<" "<<"|"<<setw(4)<<" "
<<"|"<<setw(8)<<" "<<"|"<<setw(11)
   <<" "<<"|"<<setw(11)<<"ДО 30"<<"|"<<setw(11)<<"ДО 40"
<<"|"<<setw(11)<<" "<<"|"<<endl;
   cout<< " |"<<setw(12)<<" "<<"|"<<setw(4)<<" "<<"|"<<setw(8)
   <<" "<<"|"<<setw(40)
<<"_____"
   <<"|"<<endl;
   cout<<" |"<<setw(12)<<" "<<"|"<<setw(4)<<" " <<"|"<<setw(8)
<<" "<<"|"<<setw(5)<<"Кіль-"<<"|"<<setw(5)
   <<"Від-"<<"|"<<setw(5)<<"Кіль-" <<"|"<<setw(5)
   <<"Від-"<<"|"<<setw(5)<<"Кіль-"<<"|"<<setw(5)<<"Від-"
<<"|"<<setw(5)<<"Кіль-"<<"|"<<setw(5)<<"Від-"<<"|"<<endl;
   cout<<" |"<<setw(12)<<" "<<"|"<<setw(4)<<" "<<"|"<<setw(8)
   <<" "<<"|"<<setw(5)<<"кість"<<"|"<<setw(5)<<"соток"
<<"|"<<setw(5)<<"кість"<<"|"
   <<setw(5)<< "соток"<<" "<<setw(5)<<"кість"
   <<"|"<<setw(5)<<"соток"<<"|"<<setw(5)<<"кість"<<"|"
<<setw(5)<<"соток"<<"|"<<endl;
   cout<<"_____"
       <<"_____"
   cout<<"_____" <<endl;
   cout<<" |"<<setw(12)<<" "<<"|"<<setw(4)<<" "<<"|"<<setw(8)
   <<" "<<"|"<<setw(5)<<tl<<"|"<<setw(5)<<tlpr<<"|"<<setw(5)
<<t2<<"|"<<setw(5)<<t2pr<<"|"
   <<setw(5)<<t3<<"|"<<setw(5)<<t3pr<<"|"<<setw(5)<<t4<<"|"
<<setw(5)<<t4pr<<"|"<<endl;
   cout<<"_____"
       <<"_____"
   cout<<"_____"
   <<endl;
   return 0;
   }
```


ПРИКЛАДИ АЛГОРИТМІВ ОБРОБКИ СОЦІАЛЬНО-ЕКОНОМІЧНОЇ ІНФОРМАЦІЇ

Приклад 1 Побудувати алгоритм створення файла даних, який містить таку інформацію: код підприємства, код цеху, табельний номер працівника, стаж роботи загальний, стаж роботи безперервний, розряд. Під час введення здійснювати синтаксичний контроль даних, що вводяться (рис. 1.1).

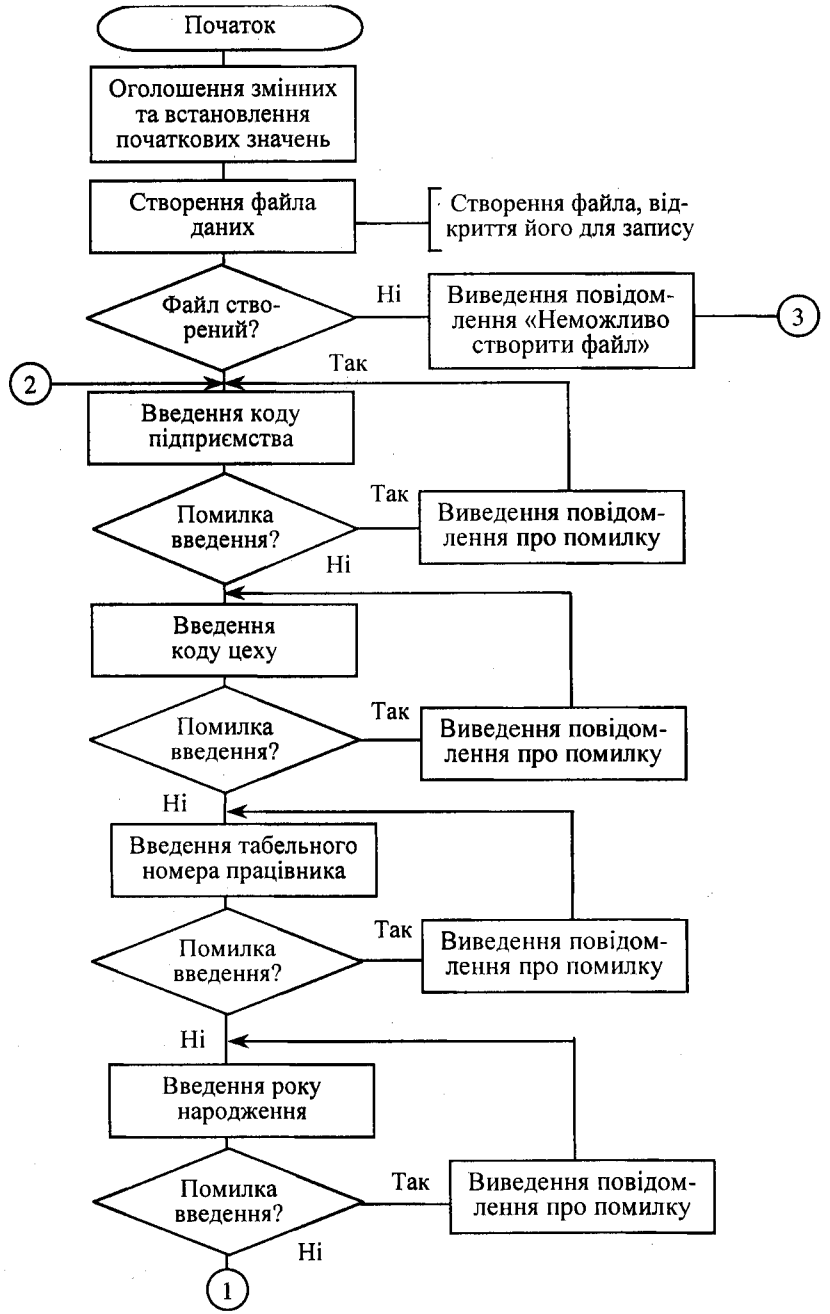
Приклад 2 На основі файла, створеного в прикладі 1, побудувати алгоритм доповнення файла даних новими записами з синтаксичним контролем (рис. 1.2).

Приклад 3 Побудувати алгоритм коригування файла, створеного в прикладі 1. Під час введення нових даних здійснювати синтаксичний контроль (рис. 1.3).

Приклад 4 Побудувати алгоритм вилучення запису з файла, створеного в прикладі 1 (рис. 1.4).

Приклад 5 Побудувати алгоритм підрахунку кількості працівників, які мають безперервний загальний стаж роботи до 5 років, від 5 до 10 років, від 10 до 20 і більше років. Використовувати файл, створений у прикладі 1 (рис. 1.5).

Приклад 6 Використовуючи файл з прикладу 1, побудувати алгоритм обробки запиту про кількість працівників заданого підприємства, що мають безперервний загальний стаж роботи до 5 років, від 5 до 10 років, від 10 до 20 і більше років (рис. 1.6).



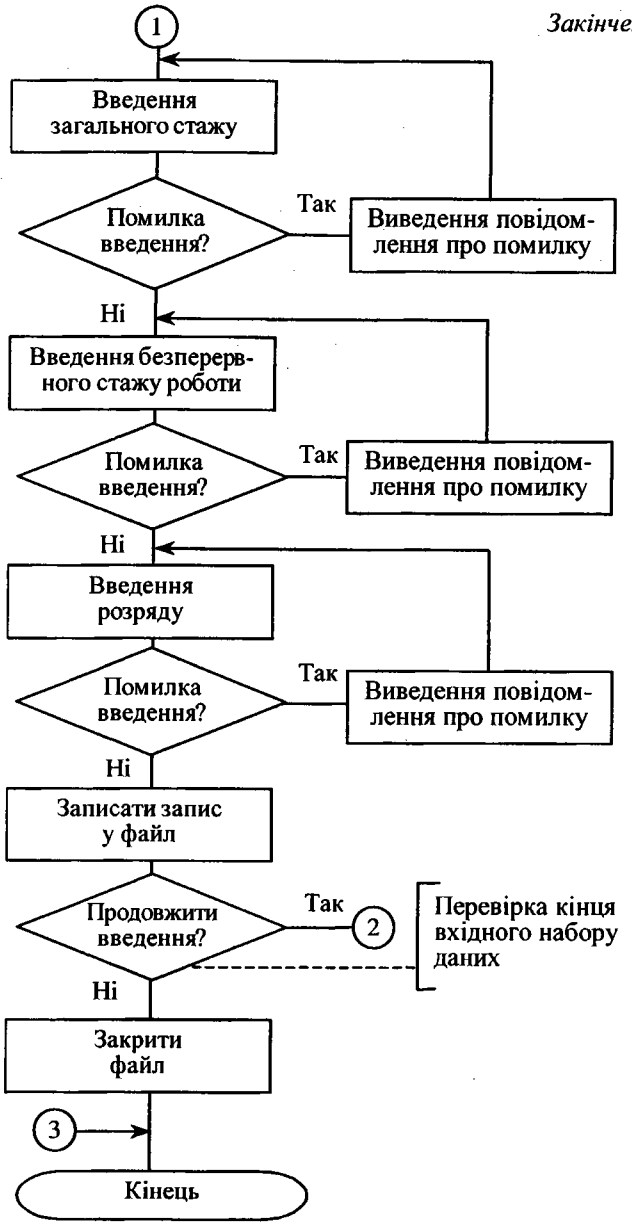


Рис. 1.1. Схема алгоритму створення файла даних з синтаксичним контролем

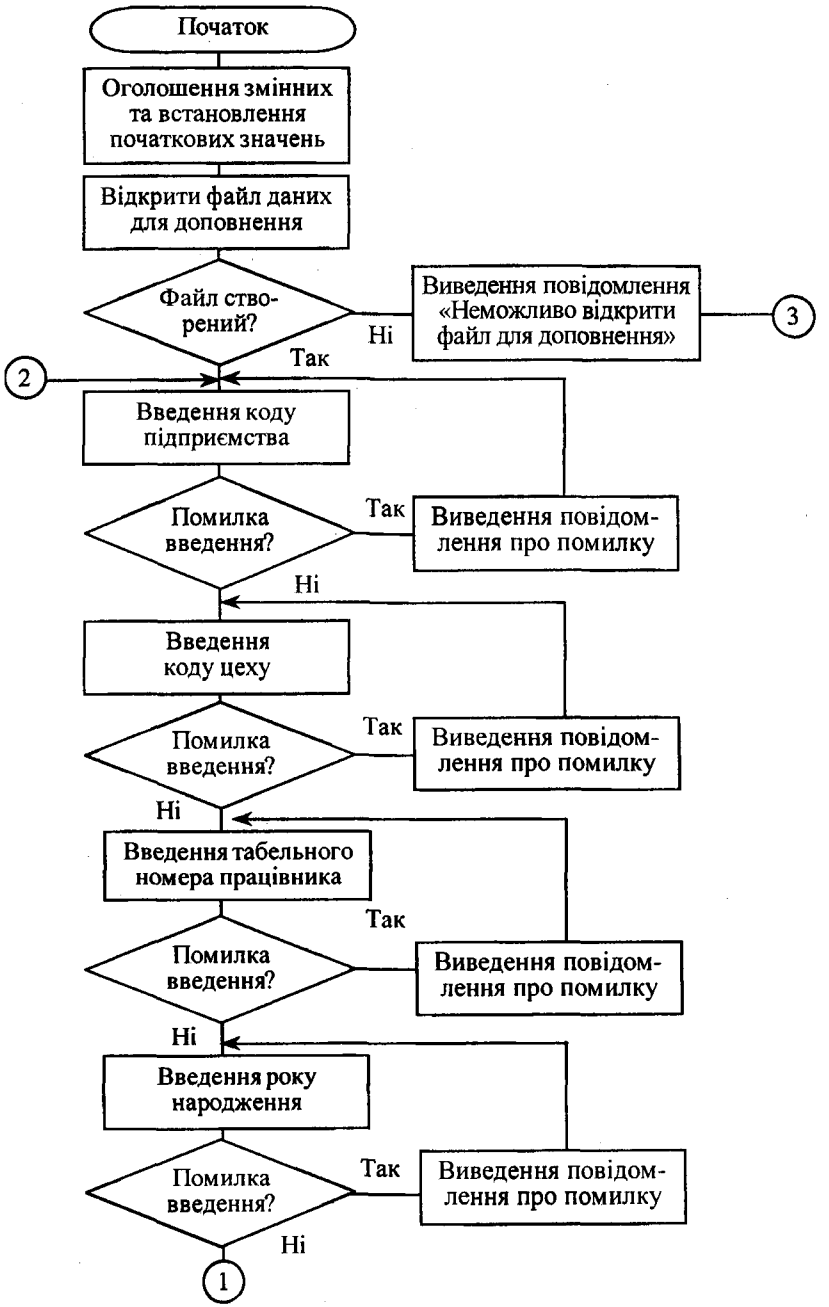
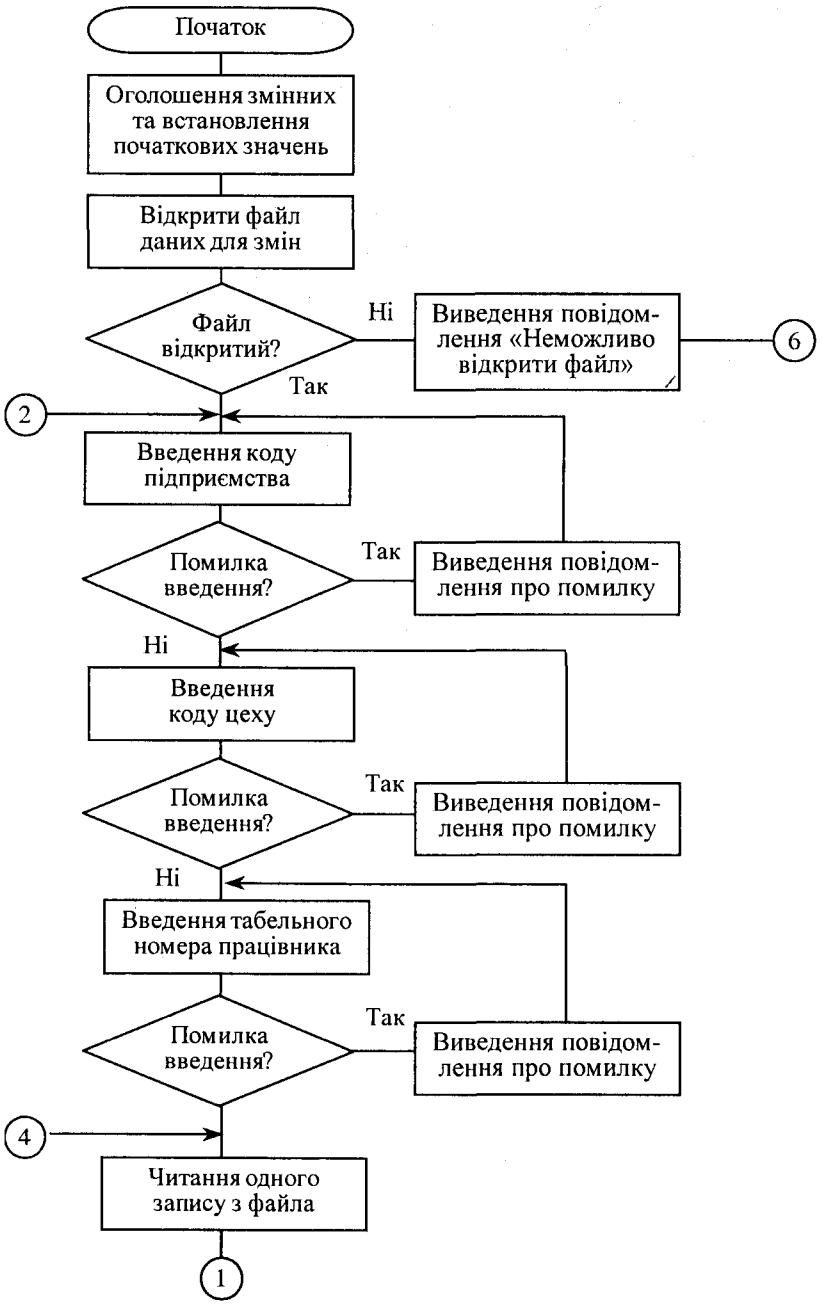
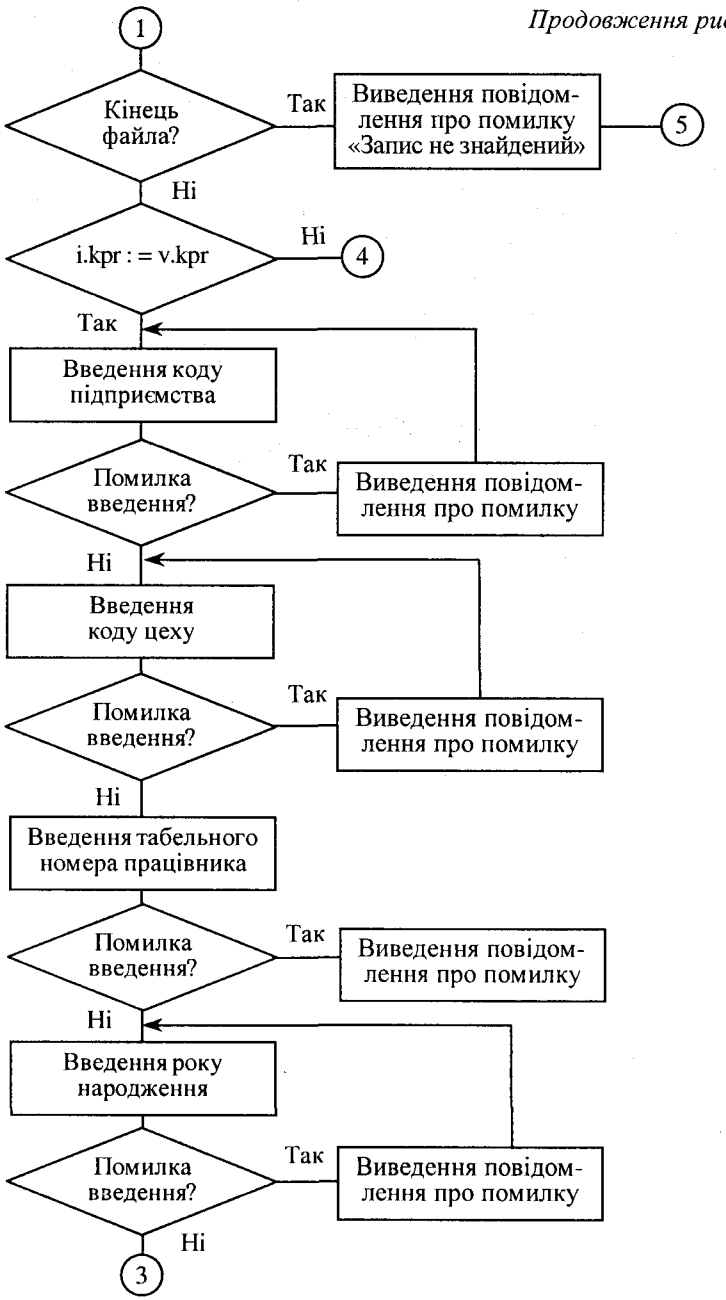


Рис. 1.2. Схема алгоритму доповнення файла даних з синтаксичним контролем



Продовження рис. 1.3



Закінчення рис. 1.3

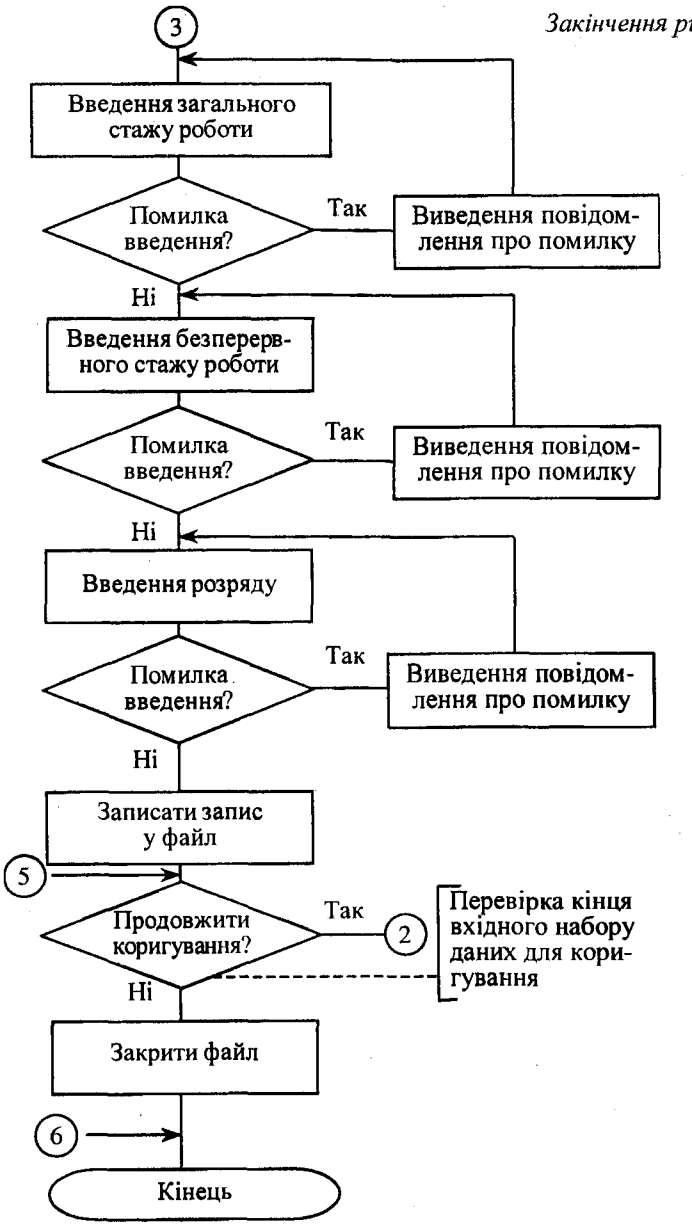


Рис. 1.3. Схема алгоритму коригування файла даних з синтаксичним контролем

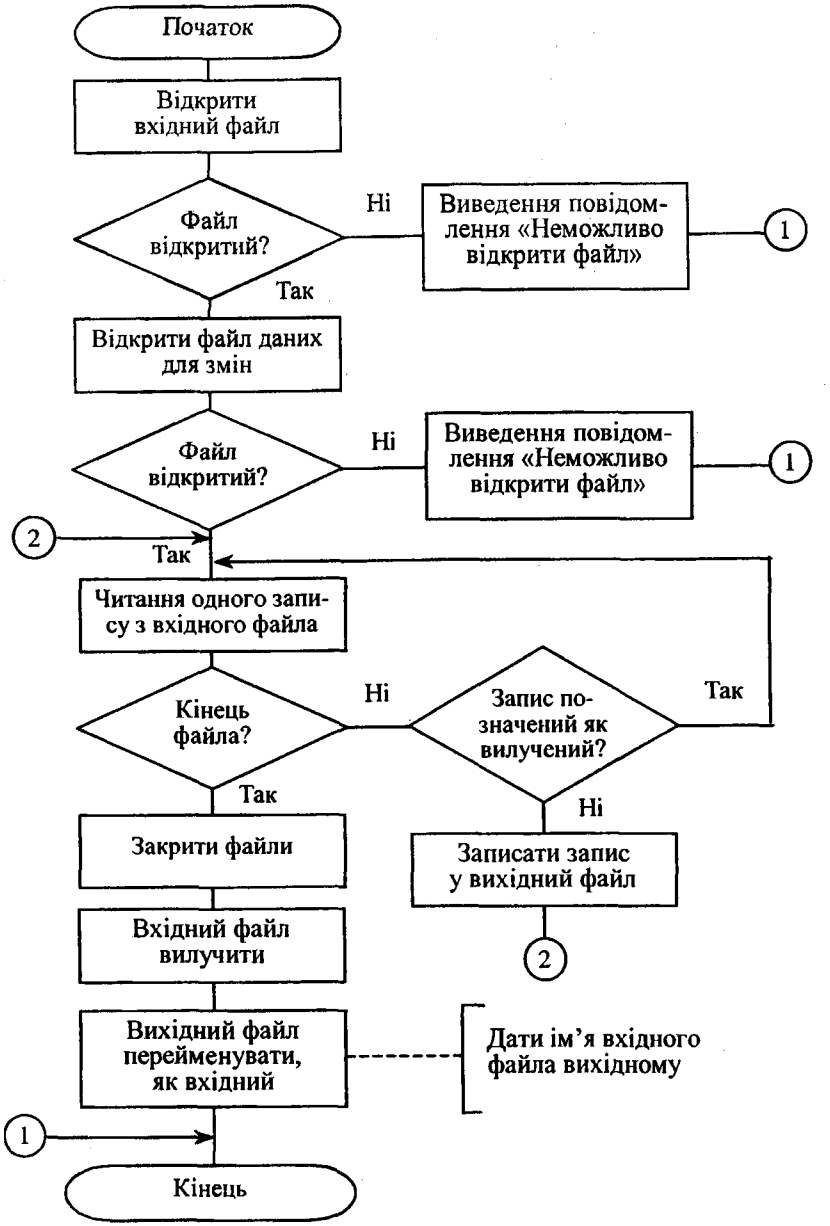
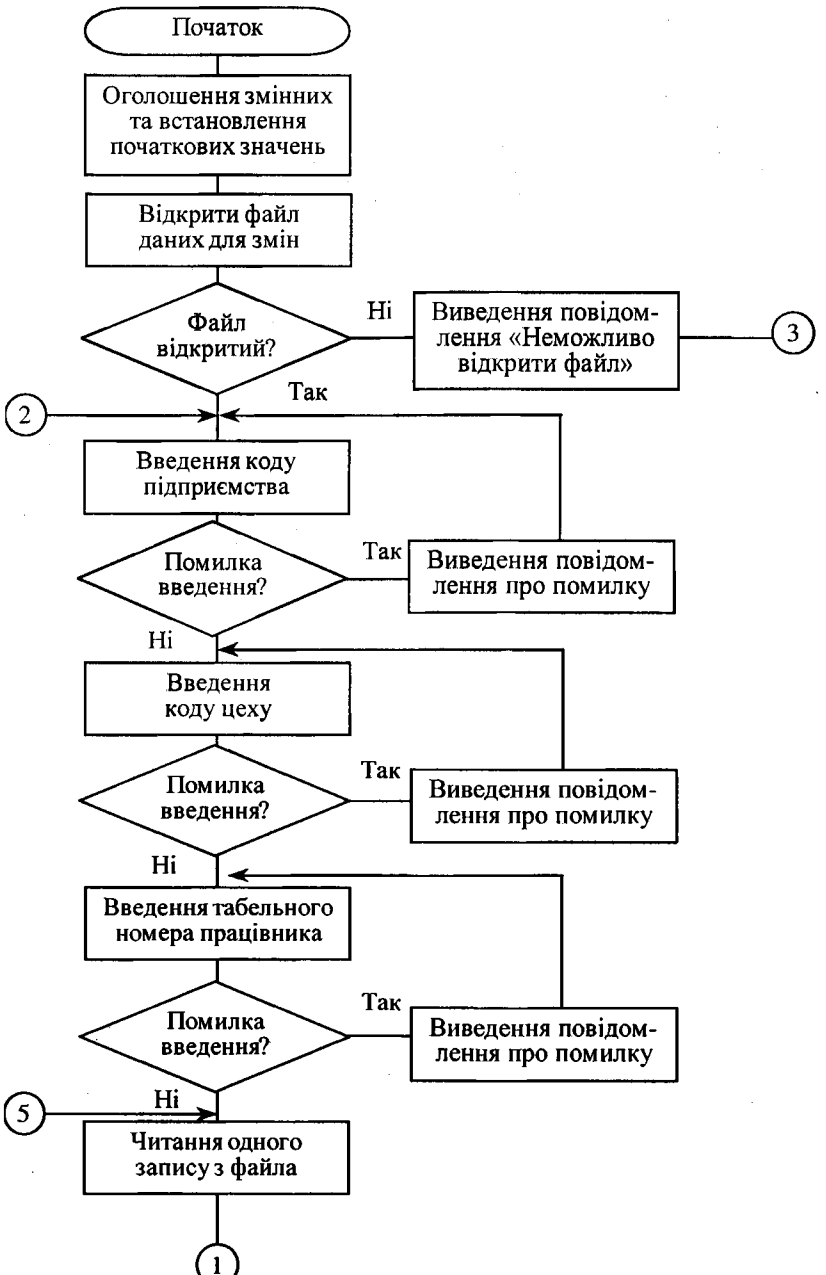


Рис. 1.4. Схема функції вилучення записів (очищення файла від вилучених записів)



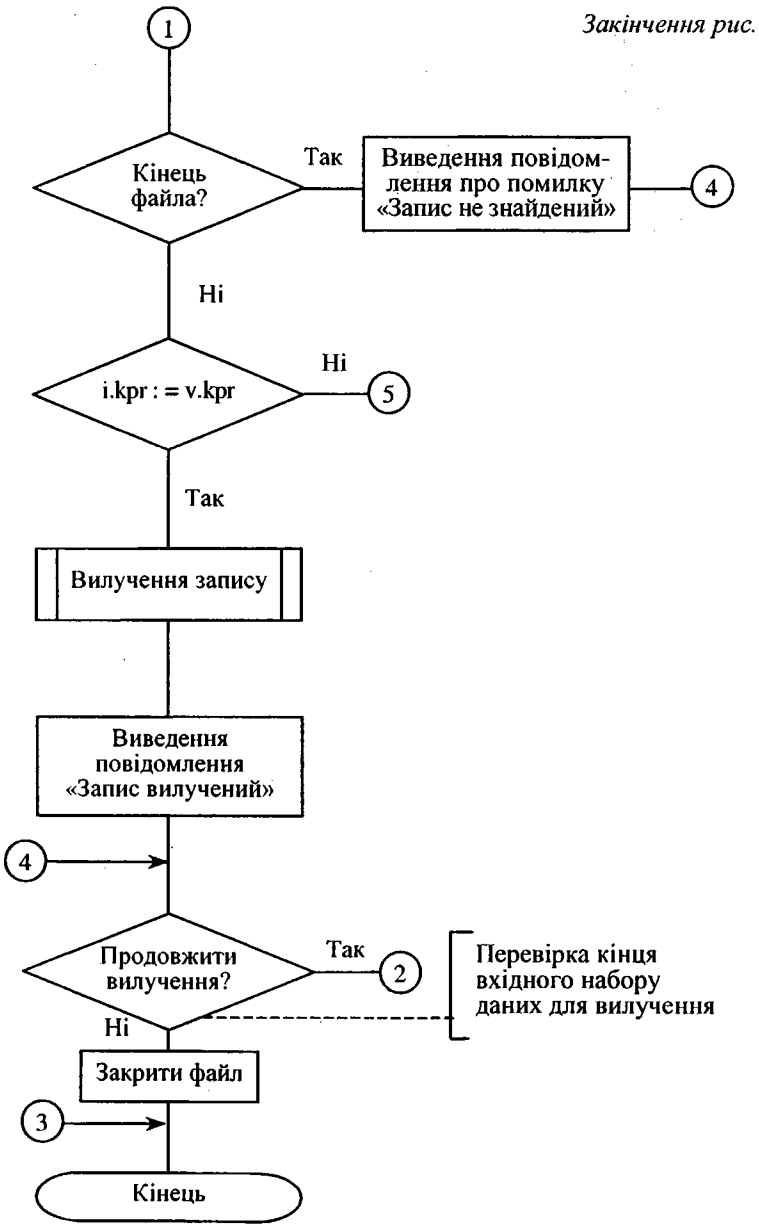
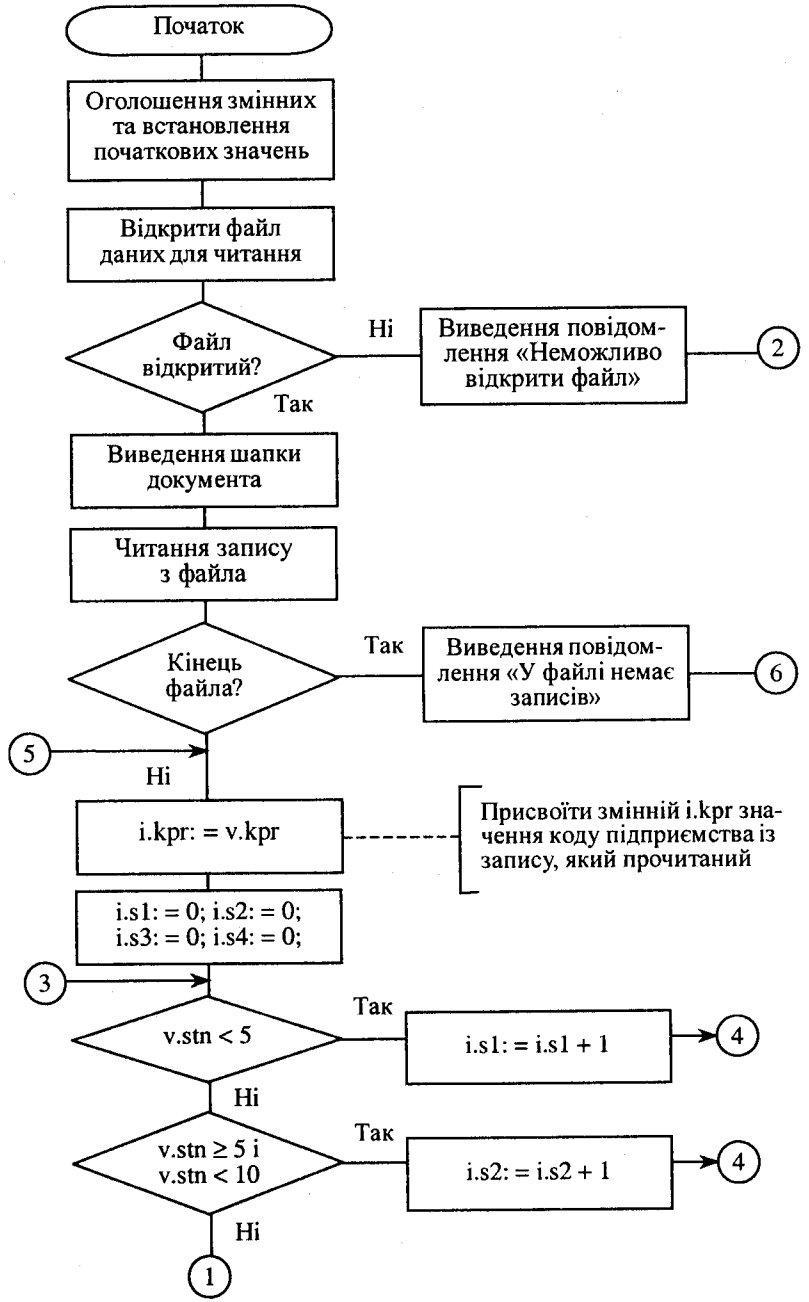


Рис. 1.5. Схема алгоритму вилучення записів з файла даних



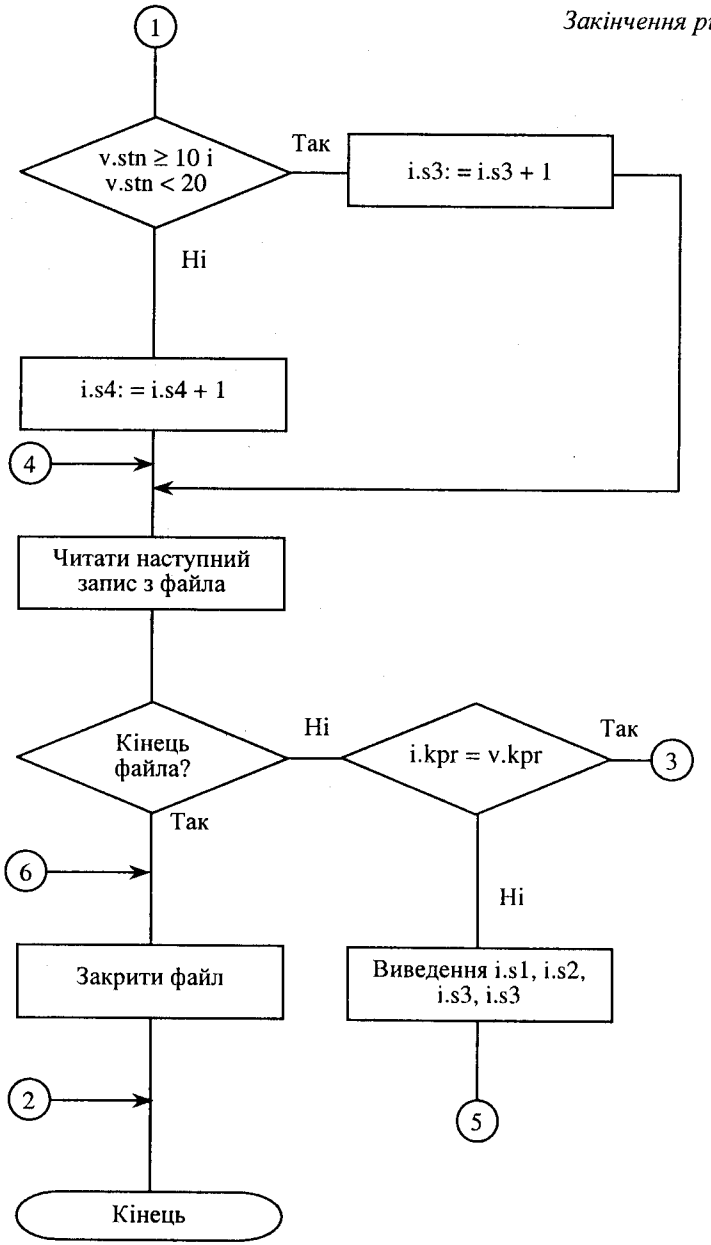
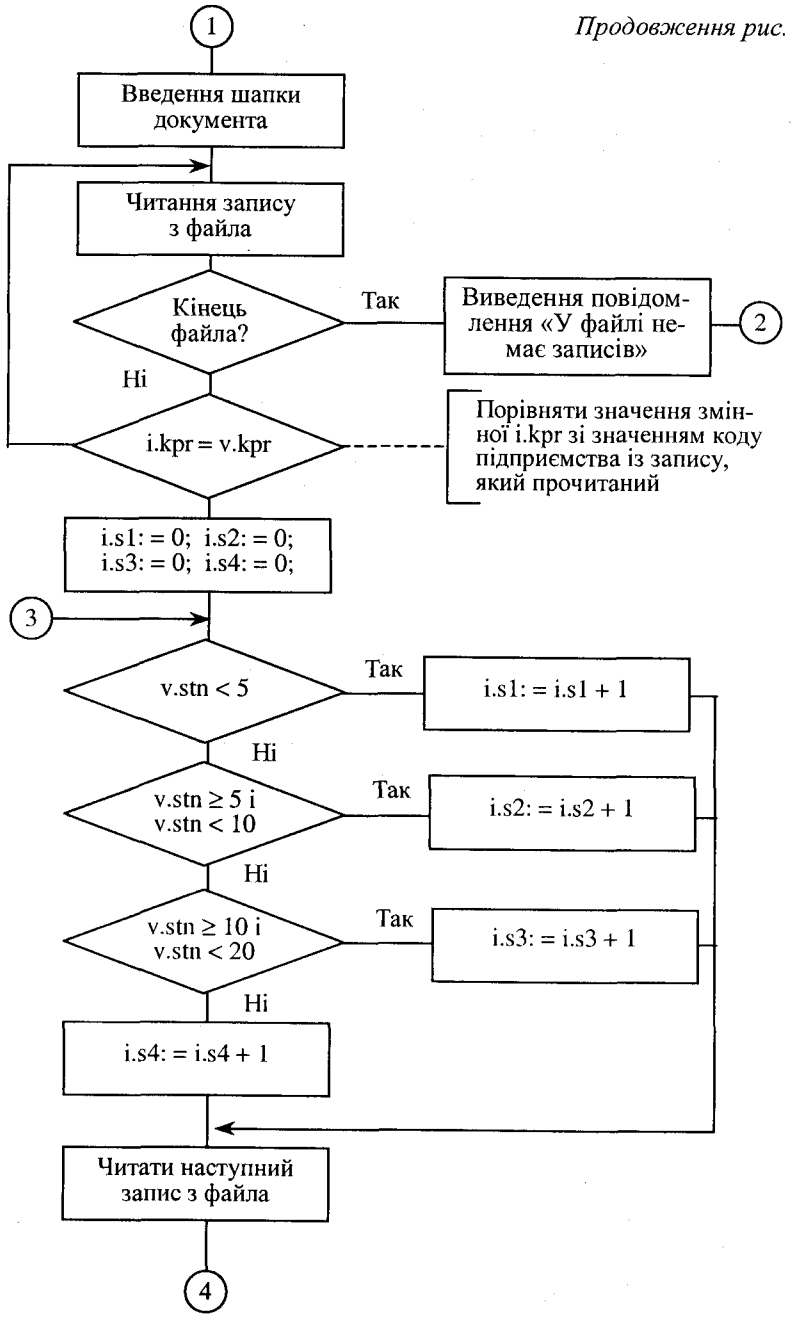


Рис. 1.6. Схема алгоритму обробки файла


Початок
Оголошення змінних та встановлення початкових значень
Відкрити файл даних для читання
Файл відкритий?
Ні
Виведення повідомлення «Неможливо відкрити файл»
5
Так
Виведення коду підприємства
Помилка введення?
Так
Виведення повідомлення про помилку
Ні
Виведення коду цеху
Помилка введення?
Так
Виведення повідомлення про помилку
Ні
Виведення табельного номера працівника
Помилка введення?
Так
Виведення повідомлення про помилку
Ні
1


1
Введення шапки документа
Читання запису з файла
Кінець файла?
Так
Виведення повідомлення «У файлі немає записів»
2
Ні
i.kpr = v.kpr
Порівняти значення змінної i.kpr зі значенням коду підприємства із запису, який прочитаний
i.s1: = 0; i.s2: = 0;
i.s3: = 0; i.s4: = 0;
3
v.stn < 5
Так
i.s1: = i.s1 + 1
Ні
v.stn ≥ 5 і v.stn < 10
Так
i.s2: = i.s2 + 1
Ні
v.stn ≥ 10 і v.stn < 20
Так
i.s3: = i.s3 + 1
Ні
i.s4: = i.s4 + 1
Читати наступний запис з файла
4

ЗМІСТ