Тарас РУДИЙ
Ярослав ПАРАНЧУК
Володимир СЕНИК

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

ЧАСТИНА 1

«Структурне програмування»

Навчальний посібник

Львів
2023

УДК 81:004.93
А45

Рекомендовано до друку та розміщення в електронних сервісах ЛьвДУВС
Вченою радою Львівського державного університету внутрішніх справ
(протокол від 25 січня 2023 року № 7)
Рекомендовано до опублікування Науково-методичною радою
Національного університету «Львівська політехніка»
(протокол від 22 лютого 2023 року № 67)

Р е ц е н з е н т и:
Соколовський Я. І. – доктор технічних наук, професор;
Крошний І. М. – кандидат технічних наук, доцент;
Кулешник Я. Ф. – кандидат технічних наук, доцент

Рудий Т. В., Паранчук Я. С., Сеник В. В.
А45 Алгоритмізація та програмування. Частина 1. Структурне програмування : навчальний посібник. Львів : Львівський державний університет внутрішніх справ, 2023. 240 с.

ISBN 978-617-511-373-8

Подано матеріали для набуття теоретичних знань основних понять алгоритмізації обчислювальних процесів і техніки застосування у програмуванні класичних базових алгоритмічних структур (організація програм) та базових структур даних (організація даних), а також практичних навичок проєктування алгоритмів та їх програмного реалізування мовою програмування С++ із застосуванням інтегрованого середовища Microsoft Visual Studio. Теоретичний матеріал доповнено великою кількістю прикладів.

Для здобувачів першого (бакалаврського) рівня вищої освіти спеціальностей: 126 "Інформаційні системи та технології", 141 "Електроенергетика, електротехніка та електромеханіка", 152 "Метрологія та інформаційно-вимірювальна техніка", а також усіх охочих освоїти мову програмування С++.

The "Algorithmization and Programming Guide: Part 1. Structural Programming" contains materials for acquiring theoretical knowledge about the basic concepts of the algorithmization of computing processes and the technique of using classic basic algorithmic structures (organization of programs) and basic data structures (organization of data) in programming, as well as practical skills in designing algorithms and their software implementation in the C++ programming language using the integrated Microsoft Visual Studio environment. Presentation of the theoretical material is accompanied by quite a large number of examples.

For applicants of the first (bachelor) level of higher education of the educational and professional program "Law enforcement information systems" specialty 126 "Information systems and technologies" and all those who wish to learn the C++ programming language.

УДК 81:004.93

© Рудий Т. В., Паранчук Я. С.,
Сеник В. В., 2023
© Львівський державний університет
внутрішніх справ, 2023

ISBN 978-617-511-373-8

ЗМІСТ

ВСТУП……………………………………………………………………..……..7

Розділ 1. АЛГОРИТМІЗАЦІЯ
ОБИСЛЮВАЛЬНИХ ПРОЦЕСІВ…………………………………………......13
1.1. Форми та засоби подання алгоритмів……………..…………14
1.2. Загальні поняття про алгоритмізацію……………………..15
1.3. Базові конструкції алгоритмів………..………………………16
1.4. Подання алгоритмів…………………………………………...18
1.5. Основні керуючі структури алгоритмів……………………...21
1.6. Лінійні алгоритми……………….……………………………….23
1.7. Розгалужені алгоритми…………….………………………….…24
1.8. Алгоритми циклічної структури……………………..…27
1.9. Приклади проєктування алгоритмів
розв’язання класичних типів задач…….………………….…...29

Розділ 2. ПАРАДИГМИ ПРОГРАМУВАННЯ……….………………….…38
2.1. Парадигми і мови програмування……………………………38
2.2. Структурне програмування…………….……………………….…43
2.2.1. Структурний підхід до проєктування
програм………………………………………………..…43
2.2.2. Принцип поділу програми на окремі
модулі…………………………………………………….45
2.2.3. Методологія покрокового деталізування
програми……………………………………………….....47
2.3. Процедурне програмування………….……………………49
2.4. Об’єктне (модульне) програмування……………………...50
2.5. Об’єктно-орієнтовне програмування……………………………..50
2.5.1. Основні ознаки об’єктно-орієнтованого
програмування………………………………………...52

2.5.2. Поняття про механізм реалізування програм методом інкапсуляції……………………….…………53
2.5.3. Поняття про властивість поліморфізму………………….….54
2.5.4. Поняття про використання процесу успадкування……………………………………………….56
2.6. Програмні середовища……………………………………….56

Розділ 3. БАЗОВІ ЕЛЕМЕНТИ С++……………………………………...65
3.1. Склад С++…………………………………………………………...65
3.2. Літерали та їхні типи…………………………………………….…66
3.2.1. Числові літерали……………………………………………67
3.2.2. Символьні літерали……………………………….………..68
3.3. Константи та змінні…………………………………….……………70
3.3.1. Константи……………………………………………….……..70
3.3.2. Змінні…………………………………………………….……75
3.3.3. Вирази та оператори………………………………….……82
3.3.4. Ключові слова………………………..………………….……83
3.3.5. Коментарі…………………………………………….…….……85
3.4. Концепція типів даних……………………………………………..85
3.5. Базові (фундаментальні) типи даних……………………….……87

Розділ 4. СТРУКТУРА С++-ПРОГРАМИ.
ОПЕРАТОРИ МОВИ С++…………………………………………………...90
4.1. Структура С++-програми……………………………………………..90
4.2. Пріоритет виконання операторів і правила асоціативності…………………………………………...96
4.3. Оператори інкременту та декременту……………………………103

Розділ 5. НАСТАНОВИ ПРИСВОЄННЯ.
ВВЕДЕННЯ/ВИВЕДЕННЯ ДАНИХ……………………………….……...106
5.1. Настанови присвоєння……………………………………….………106
5.1.1. Прості присвоєння…………………………………..…….…106
5.1.2. Складені присвоєння……………………………..…………108
5.2. Правила узгодження типів……………………………………………..109
5.3. Введення/виведення даних…………………………………………110
5.4. Поняття потоку……………………………………………………111
5.5. Обчислення арифметичних виразів…………………………….…..117

Розділ 6. СТВОРЕННЯ ПРОГРАМНИХ ПРОЄКТІВ
РОЗГАЛУЖЕНОЇ СТРУКТУРИ У С++……………………………………..125
6.1. Розгалужені алгоритми……………………………………………125
6.2. Настанова умовного переходу if
(настанова розгалуження)………………………………………..126
6.3. Приклади програмних проєктів,
які реалізовують алгоритми з розгалуженою
структурою у С++………………………………………..……135
6.4. Настанова безумовного переходу goto……………………….…139
6.5. Настанови вибору варіантів switch………………………………140
6.6. Приклади використання настанови
вибору switch……………………………………………..……..142

Розділ 7. СТВОРЕННЯ ПРОГРАМНИХ ПРОЄКТІВ
ЦИКЛІЧНОЇ СТРУКТУРИ У С++…………………………………..….149
7.1. Настанова циклу for………………………………………………149
7.2. Впорядкування (опрацювання)
послідовності чисел…………………………………..……..158
7.3. Вкладені цикли for……………………………………………..164
7.4. Настанова циклу з передумовою……………………………….167
7.5. Настанова циклу з постумовою do-while……………………..172

Розділ 8. СТРУКТУРОВАНІ ТИПИ ДАНИХ.
МАСИВИ……………………………………………………………….177
8.1. Одновимірні масиви………………………………………………177
8.2. Сортування елементів масиву……………………………………189
8.3. Організація контролю меж масивів……………………………...193
8.4. Двовимірні масиви………………………………………………..194

Розділ 9. МОДУЛЬНА ОРГАНІЗАЦІЯ ПРОГРАМ.
ОРГАНІЗАЦІЯ ФУНКЦІЙ У С++………………………………………211
9.1. Організація функцій у С++……………………………..…...213
9.1.1. Прототип функції…………………………………..213
9.1.2. Означення (реалізування) функції……………………216
9.2. Основні правила організації функцій………………………...219
9.3. Передавання параметрів у функцію……………………………..222

9.3.1. Передавання аргументів за значенням…………..……….223
9.3.2. Функції, які не повертають значення……………………224
9.3.3. Функції, які повертають значення…………..……………225
9.4. Приклади програмних кодів з використанням функцій…………………………………………………………..…226
9.4.1. Опрацювання масивів у функціях……………………….230

ВИКОРИСТАНА ЛІТЕРАТУРА…………………..……………………….236

ПРЕДМЕТНИЙ ПОКАЖЧИК……………………………………..………….237

ВСТУП

Програмування – це складний, творчий інженерний процес, неможливий без застосування відповідних технологій та теоретичних знань. Властиво, що ознайомлення з технологіями створення проєктів програмних кодів мовою програмування C++ є основним завданням посібника. Тобто навчальний посібник стосується не так мови програмування, як технології програмування цією мовою зокрема, а також практики програмування назагал.

Для програмного реалізування алгоритмів обрано мову програмування C++. Вона є високорівневою мовою загального призначення зі статичним типізуванням, яка надається для створення різноманітних застосунків. Своїм "корінням" вона сягає мови С, яку розробив в 1969–1973 роках у компанії *Bell Labs* програміст Денніс Рітчі (*Dennis Ritchie*).

На початку 1980-х років данський програміст Бьорн Страуструп (*Bjarne Stroustrup*), який тоді працював у компанії *Bell Labs*, розробив C++ як розширення до мови С. Практично спочатку C++ легко доповнювала мову С деякими можливостями об'єктно-орієнтованого програмування, тому сам Страуструп називав її як "*C with classes*" ("С з класами").

Згодом нова мова популяризувалася. До неї були додані нові можливості, які робили її не просто доповненням до С, а новою мовою програмування. У результаті "С з класами" перейменували на C++. Відтоді обидві мови почали розвиватися незалежно одна від одної.

C++ є потужною мовою, успадкувавши від С багаті можливості роботи з пам'яттю комп'ютера. Тому, нерідко, C++ застосовують у системному програмуванні, зокрема для створення операційних систем (ОС), драйверів, різних утиліт, антивірусів тощо.

До речі, ОС *Windows* здебільшого написана мовою C++. Але лише системним програмуванням застосування цієї мови не обмежується. C++ можна використовувати у програмах довільного рівня, де важливі швидкість роботи та продуктивність.

C++ є мовою, яка компілюється, а це означає, що компілятор транслює початковий програмний код на C++ у виконуваний файл, який містить набір машинних настанов. Однак різні платформи мають свої особливості, тому скомпільовані програми не можна просто перенести з однієї платформи на іншу і там уже запустити на виконання.

На рівні початкового коду програми на C++ зазвичай мають переносимість, якщо не використовуються якісь специфічні для поточної ОС функції. Наявність компіляторів, бібліотек та інструментів розроблення майже під усі поширені платформи дає змогу компілювати той самий початковий код на C++ у застосунки під ці платформи.

На відміну від C, мова C++ дає можливість створювати застосунки в об'єктно-орієнтованому стилі, подаючи програмний код як сукупність класів і об'єктів, які взаємодіють між собою. Це полегшує створення великих застосунків.

Повертаючись до питання про вибір мови для курсу "Алгоритмізація та програмування", хочемо наголосити, що автори цілковито погоджуються з думкою В. В. Бублика, який рекомендує C++ як мультипарадигменну мову, найбільш повну з огляду на концепцію сучасного програмування. Наступники C++, а саме Java і C#, наразі парадигму не змінили, зате деякі важливі риси, наявні в C++, втратили.

У навчальному посібнику подано основні поняття алгоритмізації обчислювальних процесів, програмне реалізування алгоритмів на основі знань про дані базових типів, оператори C++, настанови, підпрограми. Переконані з власного досвіду, що ґрунтовне вивчення усіх наявних можливостей мови програмування C++ та численних бібліотечних засобів не може бути предметом вступного курсу "Алгоритмізація та програмування", тому найбільшу увагу в посібнику зосереджено на викладенні основних мовних конструкцій і технологій їх застосування, а не на можливостях численних бібліотек або нюансів компіляторів. Також аналізуються елементи технології програмування: проєктування і відлагодження програм, вибір ідентифікаторів, коментування, розподіл обов'язків між частинами програмного коду, використання підпрограм.

Основне завдання цього навчального посібника – допомогти здобувачам вищої освіти навчитися розробляти проєкти програмних кодів мовою C++ із застосуванням технології структурного програмування.

Теоретичний матеріал, викладений у посібнику, поданий так, що для освоєння матеріалу достатньо шкільних знань. Подані у посібнику приклади ретельно аналізуються щодо кожного з етапів вивчення окремих розділів.

Метою цього навчального посібника є ознайомлення здобувачів вищої освіти з основами алгоритмізації обчислювальних процесів, створення проєктів програмних кодів мовою C++, спрямування останніх на індивідуальну роботу шляхом детального аналізу запропонованих прикладів програмних кодів.

Вважаємо, що у посібнику подано достатньо прикладів, необхідних для успішного засвоєння матеріалу. Такий підхід дає змогу зосередитись на самостійному вивченні та відпрацюванні основних типових прийомів програмування.

Досягненню обраної мети підпорядковано структуру навчального посібника, який складається з дев'яти розділів. Матеріал побудовано за принципом повторюваності з попереднім поясненням дій для виконання тієї чи іншої операції. Водночас пояснення подаються, якщо дія виконується вперше. Викладення наступного навчального матеріалу передбачає, що користувач вже володіє попереднім теоретичним матеріалом і уважно виконав аналіз поданих прикладів. Тому, якщо деяке формулювання, приклад, фрагмент програмного коду здаються незрозумілими, потрібно повернутися назад, до попередніх теоретичних пояснень або відповідних прикладів.

У першому розділі детально розкрито питання алгоритмізації обчислювальних процесів та подання алгоритмів, а також подано блок-схеми алгоритмів розв'язання типових задач.

Парадигми імперативного програмування і мови програмування розглянуто у другому розділі.

У третьому розділі викладено базові елементи C++. Основну увагу зосереджено на аналізі таких понять, як константи, змінні, оператори, вирази. Викладено концепцію типів даних та розглянуто фундаментальні типи даних.

Четвертий розділ присвячено формуванню загальної структури C++-програми. Викладено пріоритет виконання операторів та правила асоціативності, оператори інкременту і декременту.

У п'ятому розділі викладено теоретичні засади і подано приклади використання настанови присвоєння, поняття потоку та введення/виведення даних, подано підходи до формування й обчислення арифметичних виразів.

Шостий і сьомий розділи присвячені створенню програмних проєктів розгалуженої структури на основі використання настанов керування. Подано детальний опис та типові приклади використання останніх. Описано загальні підходи до створення програмних проєктів циклічної структури у С++. Розглянуто вкладені структури циклів та типові задачі впорядкування послідовності чисел.

Структуровані типи даних детально подані у восьмому розділі та дають змогу користувачам ознайомитися з поняттям масиву. Розглянуто задачі впорядкування елементів масивів та організацію контролю меж масивів.

У дев'ятому розділі розглядається модульна організація програм та функцій у С++. Описуються поняття прототипу та означення функції, передавання параметрів функцій, подано приклади програмних кодів з використанням функцій.

Деякі поняття, які використовуються у тексті, не означені. Це зроблено для того, щоб не переобтяжувати посібник теоретичним матеріалом, яким здобувач вищої освіти повинен володіти, прослухавши лекційний курс.

Звичайно, у межах навчального посібника теоретичний матеріал та приклади програмних кодів не можуть охопити всю повноту можливостей С++. Проте, набутих навичок буде достатньо для вільного володіння С++ у межах початківця, використання цих засобів для розв'язання типових задач та для подальшої самостійної роботи.

Передбачається використання посібника на лабораторних заняттях і для самостійної роботи студентів. Він містить численні приклади програм, функцій і окремих фрагментів коду, які наочно ілюструють викладений теоретичний матеріал та привчають до "читання" (аналізу і розуміння) чужих програмних кодів, що є обов'язковим етапом перед проєктуванням власних. Важливо, щоб студент, працюючи з програмним кодом і прикладами посібника, мав можливість відразу створювати й запускати пропоновані програми на виконання з подальшим аналізом отриманих результатів на рівні чисел. Подані у посібнику задачі є мініпроєктами для самостійного розроблення.

Поважною виявилася проблема застосування у навчальному процесі української термінології. У царині програмування вона, на жаль, ще не сформувалася остаточно. Доступні книги іншомовні, зокрема, найпоширеніші написані чужоземними мовами. Деякі терміни ще чекають на влучні переклади, інші перекладають дослівно навіть у тих випадках, коли можна вживати більш звичні та зрозумілі слова.

На державному рівні науковці ще не встигли остаточно визначити мовну норму, а радше розпочали експеримент, тому проблема термінології ще чекає на своє розв'язання.

Використання національної термінології у сфері інформаційних технологій (IT) є надзвичайно важливим. Які терміни навчимо використовувати першокурсника і які терміни використовує викладач, такі майбутній фахівець і буде використовувати у повсякденній роботі.

Зокрема, в українській наково-технічній термінології є певна плутанина щодо перекладу і трактування англійського терміну *"statement"* – доволі часто бачимо термін "оператор", який з'явився у 1965 році під час перекладу. Перекладаючи *"Revised Report on the Algorithmic Language Algol 60"*, А. П. Єршов та М. Р. Шура-Бура переклали англійське слово *"operator"* як "знак операції", а *"statement"* – як "оператор", хоча в англомовній термінології *"operator"* (+, -, % тощо) слугує для позначення операцій над операндами, що зумовило чимало неузгодженостей у вітчизняній науці.

Також для позначення інструкцій (присвоєння, умовного переходу тощо), які в англомовній науково-технічній літературі однозначно позначаються терміном *"statement"*, у вітчизняній літературі низка перекладачів вживають оператор, інколи вираз, інколи команда. Для позначення складених інструкцій (if, if-else, while та ін.) у вітчизняних джерелах трапляється термін "конструкція" (наприклад, "конструкція while").

Для початку з'ясуємо, як будемо тлумачити термін "оператор" у межах цього посібника. Опера́тор (*operator*) – спеціальний символ, який надає вказівку компілятору про те, яку потрібно виконати дію з деякими операндами (наприклад, +, –, %, << тощо).

Зазвичай, мови програмування мають набір операторів, схожих до операторів у математиці: у певному розумінні, оператори є спеціальними функціями.

На відміну від функцій, оператори є основними конструкціями мови програмування C++, їх позначення коротші та містять спеціальні символи.

Інколи під оператором розуміють операцію, хоча правильніше казати, що оператор вказує на те, яку операцію потрібно виконати.

До того ж в українській технічній літературі "операторами" називають окремі види інструкцій, такі як цикли й умовні інструкції.

Цьому також сприяли відмінності в термінології різних мов програмування, особливо ранніх.

У межах цього навчального посібника автори для позначення англомовного терміну *"statement"* пропонують вживати національний термін ***"настанова"*** з огляду на те, що термін "інструкція" є дослівним перекладом англійського слова *"instruction"* (трактується як: інструктаж, інструкція, дозвіл, навід, навчання, направа). Слово "настанова" в українській мові трактується як: вказівка, припис, директива, інструкція, рекомендація. На нашу думку, термін "настанова" більш повно і, головне, зрозуміліше відтворює суть англійського слова (терміна) *"statement"*. Автори не запроваджують новий термін, а повністю підтримують доктора технічних наук, професора Юрія Грицюка та доктора технічних наук, професора Тараса Рака, які вперше запропонували використовувати термін *"настанова"* у 2011 році.

Навчальний осібник рекомендований найперше для здобувачів першого (бакалаврського) рівня вищої освіти спеціальностей: 126 "Інформаційні системи та технології", 141 "Електроенергетика, електротехніка та електромеханіка", 152 "Метрологія та інформаційно-вимірювальна техніка", а також усіх охочих освоїти мову програмування C++.

Автори висловлюють свою вдячність і повагу студентам спеціальності 126 "Інформаційні системи та технології" першого і другого курсів факультету № 2 Львівського державного університету внутрішніх справ, які без зайвих нарікань і з неймовірним терпцем допомогли апробувати теоретичний і практичний матеріал, викладений у посібнику.

Також висловлюємо щиру подяку рецензентам та доктору технічних наук, професорові Грицюку Юрію Івановичу за співпрацю в роботі над посібником і практичні поради.

Розділ
1

АЛГОРИТМІЗАЦІЯ ОБИСЛЮВАЛЬНИХ ПРОЦЕСІВ

Алгоритм є математичною абстракцією програми. Застосування технічних пристроїв висуває дуже строгі вимоги до точності опису правил і послідовності виконання дій. Алгоритм можна розглядати як певний універсальний засіб для розв'язання цілого класу задач.

Перехід від інтуїтивного поняття алгоритму до нормального визначення алгоритму призвів до того, що об'єктами оброблення стали самі алгоритми. Основними завданнями алгоритмізації є автоматичне верифікування і оптимізування програм і системи з розпаралелюванням виконання програм на багатопроцесорних обчислювальних системах. Наступним напрямом є лінгвістичні алгоритми: перевірка орфографії, автоматичний переклад, програми, які "розмовляють", робота з граматикою.

Формування наукового поняття алгоритму, яке стало важливою науковою проблемою, сьогодні ще не закінчене. Хоча теорія алгоритмів є математичною дисципліною, вона ще не дуже схожа на такі загальновідомі науки, як геометрія або теорія чисел. Вона ще тільки зароджується, причому тим початковим матеріалом, на підставі якого повинно бути побудоване широке наукове поняття алгоритму, є інтуїтивне поняття, також дуже широке, але недостатньо ясне.

У реальному житті виконання довільних дій пов'язане з витрачанням різних ресурсів: матеріалів, енергії і часу. Навіть здійснюючи довільні записи, ми витрачаємо ресурси (наприклад, папір, чорнило і час).

Ще недавно деякі задачі не можна було розв'язати через надто велику кількість необхідних для цього операцій і надто малу швидкість їх виконання. Створення електронних обчислювальних машин зробило такі задачі розв'язуваними.

1.1. Форми та засоби подання алгоритмів

Алгоритм – це система правил, яка сформульована мовою, зрозумілою виконавцеві алгоритму, визначає процес переходу від допустимих початкових даних до певного результату і володіє властивостями масовості, скінченності, визначеності, детермінованості. Це визначення алгоритму не є строгим (хоча б тому, що в ньому використовуються не точно визначені терміни).

Протягом багатьох століть поняття алгоритму пов'язувалося з числами і відносно простими діями над ними. Так, властиво, і математика була наукою про обчислення, наукою прикладною. Найчастіше алгоритми подавалися у вигляді математичних формул. Порядок елементарних кроків алгоритму задавався розставленням дужок, а кроки полягали у виконанні арифметичних операцій і операцій відношення (перевірки рівності, нерівності тощо). Хоча ці формули могли бути достатньо громіздкими, а ручні обчислення – вкрай трудомісткими (французький астроном Жан Жозеф Левер'є витратив десятки років розраховуючи орбіти планет Сонячної системи, в результаті чого йому поталанило виявити невідому раніше планету Нептун), суть самого обчислювального процесу залишалася цілком очевидною.

У математиків не виникало потреби в усвідомленні та строгому визначенні поняття алгоритму, у його узагальненні. Але з розвитком математики з'являлися нові об'єкти, якими доводилося оперувати ученим: вектори, матриці, графи, множини тощо. Як визначити для них однозначність або як встановити скінченність алгоритму, які кроки можуть вважатися елементарними (наприклад, чи є таким обернення матриці або знаходження перетинання двох множин)? Виникла ідея про існування алгоритмічно

нерозв’язуваних проблем, таких, для яких неможливо визначити процедуру розв’язку. Отже, потрібно було навчитися математично строго доводити факт відсутності відповідного алгоритму. Це можна зробити тільки у разі створення строгого визначення алгоритму.

Спроби виробити таке визначення і призвели до виникнення теорії алгоритмів у 20–30-х рр. XX ст.

1.2. Загальні поняття про алгоритмізацію

Як уже було попередньо з’ясовано, алгоритм описує розв’язання задачі у вигляді точно визначеної послідовності дій для певного виконавця з перетворення початкових даних у результати. Оскільки однієї мети можна досягти різними способами (за рахунок різних зусиль і витрат), тому, відповідно, і алгоритмів досягнення цієї мети можна побудувати багато.

Процес побудови алгоритмів називають алгоритмізацією. Алгоритм повинен містити необхідну і достатню інформацію про перебіг здійснення процесу розв’язання конкретної задачі (проблеми). Один і той же процес можна планувати і розписувати з різним ступенем деталізування. Наприклад, директора заводу може цікавити, як здійснюється виробництво продукції на рівні взаємодії цехів, бригадира ж конкретного цеху більше цікавить, чим займається кожен робітник його бригади. Зайве деталізування алгоритму для директора буде заважати у керівництві заводом, як і нестача інформації про робітників, не дозволить бригадиру організувати ефективну роботу бригади.

Очевидно також, що певний виконавець може сприймати алгоритм, виконувати його, тільки тоді, коли алгоритм поданий у зрозумілому йому вигляді. Це означає, що довільне подання алгоритму є певним інформаційним блоком, тобто подання алгоритму є інформацією і вона розташована на певному носії інформації. Отож сам алгоритм є певною абстракцією, але реальне його реалізування можливе тільки у вигляді його подання.

Очевидно також, що у довільного алгоритму можуть бути різні форми подання. Вони можуть бути схожими, однак можуть і не мати нічого спільного, крім реалізування одного алгоритму. В інформатиці склалися цілком визначені традиції у поданні алгоритмів розрахованих на різних виконавців. Засоби, які використовуються для запису алгоритмів значною мірою визначаються тим, для якого виконавця призначається алгоритм. Але довільна форма запису (подання) алгоритму повинна забезпечувати властивості алгоритму: дискретність, детермінованість, масовість, скінченність та зрозумілість.

Вибір способу запису алгоритму залежить і від характеру задачі. Алгоритм обчислювального процесу можна записати формулою (послідовністю формул) або вербально (словами, природною людською мовою). Зрозуміло, що алгоритм заварювання чаю мабуть зручніше записати вербально, у пронумерованих пунктах, а ось алгоритм розв'язання квадратного рівняння буде більш зрозумілим під час його запису комбінацією слів і формул.

Якщо з алгоритмом працює людина, тоді це може бути й традиційна людська мова (англійська, японська або українська), мова малюнків, мова символів тощо. Однак доволі часто застосування цих природних засобів спілкування, призводить до непорозуміння. Іноді причина криється в неоднозначності використовуваної термінології. Наприклад, речення: "Відвідування онуків – це велике навантаження на нервову систему" може мати подвійний зміст: або приїзд онуків викликає дуже багато турбот, або поїздка до них є серйозним іспитом для людини похилого віку. Друге джерело проблем – це неправильне розуміння алгоритму, яке викликане недостатньою деталізацією його опису.

1.3. Базові конструкції алгоритмів

У теорії програмування доведено, що програму для розв'язання задачі довільної складності можна сформувати тільки з трьох структур, які називають лінійністю, розгалуженням і циклом. Цей результат встановлений Боймом і Якобіні ще у 1966 році шляхом

доведення того, що довільну програму можна перетворити у еквівалентну, яка складається зі згаданих структур і їх комбінацій.

Особливістю базових конструкцій є те, що довільна з них має тільки один вхід і один вихід, тому конструкції можуть бути вкладеними одна в одну довільно.

Алгоритм є основним поняттям інформатики. Відомий середньоазіатський мудрець, вчений, філософ і математик Мухаммед бен Муса аль-Хорезмі у IX ст. детально розробив правила чотирьох арифметичних дій (їх можна назвати алгоритмами арифметичних дій). Після перекладу його наукових трактатів вперше з'явився термін "**алгоритм**" (*аль-Хорезмі – Algorithmi*).

Алгоритм – це наперед заданий чіткий опис скінченної послідовності вказівок (команд), виконання яких дає змогу одержати правильний розв'язок задачі.

Алгоритм – це набір інструкцій, що описує, як деяке завдання може бути виконане. Інакше кажучи, алгоритм – це система формальних правил, яка визначає зміст і порядок дій над початковими даними і проміжними результатами, необхідними для отримання кінцевого результату під час розв'язування задачі.

Алгоритм повинен відповідати певним вимогам і мати такі властивості:

– *визначеність* (детермінованість) – алгоритм має бути чітким і однозначним, кожна команда не повинна допускати довільності тлумачення, кожний крок алгоритму має бути точно визначеним;

– *масовість* – алгоритм повинен бути, за можливості, універсальним, розрахованим на розв'язання однотипних задач з різними початковими даними;

– *дискретність* – визначений алгоритмом обчислювальний процес повинен мати дискретний (перервний) характер, тобто подаватися послідовністю окремих завершених кроків – команд або дій;

– *результативність* – кожна дія має приводити до певного результату;

– *формальність* – довільний виконавець, діючи за алгоритмом, може реалізувати коректно поставлене завдання;

– *скінченність* – розв'язок задачі з використанням алгоритму має бути одержаним за скінченну кількість кроків.

Розроблення алгоритму трохи складнішої задачі вимагає високої кваліфікації виконавця і розуміння фізичного змісту задачі. З реалізуванням алгоритму безпосередньо пов'язане вміння застосувати цей алгоритм до конкретних початкових даних розв'язуваної задачі. Таке застосування називається алгоритмічним процесом.

Цей процес полягає у перетворенні початкових даних за правилами, визначеними заданим або розробленим алгоритмом. Алгоритмічний процес загалом складається із самостійних етапів, кожен з яких призначений для переведення даних з одного стану до іншого. Одним із завдань кожного етапу обчислень є також визначення свого наступника. З поняттям алгоритмічного процесу пов'язане і поняття обчислювального процесу.

1.4. Подання алгоритмів

Запис алгоритмів здійснюється при їх розробленні та поданні. Алгоритм є певною інструкцією для виконавця, яку можна задати різними способами – словами, формулами, послідовністю обчислюваних операцій або логічних дій тощо. На практиці застосовують різні способи запису алгоритмів у текстовій та графічній формі.

Велика кількість реальних завдань доволі складна, тому між словесним описом дій і програмою алгоритмічною мовою виконується проміжний етап – побудова схеми алгоритму. Цей етап є найбільш наочним способом зображення алгоритмів у вигляді графічних схем. Схема є нібито начерком структури програмного реалізування та своєрідною "дорожньою картою" за вже готовою програмою. При графічному способі зображення алгоритмів потрібно виконувати вимоги стандарту.

Блок-схеми – це графічне зображення логічної структури алгоритму, у якому кожний етап процесу оброблення даних подається у вигляді геометричних фігур (функціональних блоків), які мають певну конфігурацію залежно від характеру виконуваних операцій.

Цікаво, що блок-схему, яку більшість з нас зазвичай уявляє, коли чує слово "алгоритм", придумав математик Джон фон Нейман, який ілюстрував блок-схемою модель зіткнення ядерних частинок при розробленні Мангетенського проєкту.

Переваги графічного способу подання алгоритмів були очевидні. Завдяки своїй компактності та наочності, подання алгоритму у вигляді взаємопов’язаних функціональних блоків, які відповідають за виконання певних дій, набуло поширення. Спочатку кожний з великих виробників комп’ютерів розробляв свою систему блоків, які відтворювали його підхід до оброблення інформації.

Намагаючись перевершити конкурентів, компанії навіть випускали власні трафарети для креслення блок-схем алгоритмів, які тоді були більш популярними, ніж спеціалізоване програмне забезпечення, включаючи спеціально розроблену для креслення блок-схем мову SFL (*Systems Flowchart Language*).

Свої блок-схеми мали й окремі організації, наприклад, військово-повітряні сили США. У 1961 р. Міжнародна організація стандартів (ISO – *International Standards Organization*) заснувала комітет "Комп’ютери і оброблення інформації", учасниками якого були представники комп’ютерних компаній і компаній-користувачів, на який покладався обов’язок розроблення стандартів для блок-схем.

Зі своїм завданням комітет упорався у 1963 р., коли був випущений перший стандарт, де визначено усі основні умовні позначення, які стосувались алгоритмів, а також правила креслення функціональних блоків.

Головне призначення стандарту – визначити зовнішній вигляд блоків та їх призначення. Для кожного з них вказується форма, але не розмір. Схеми алгоритмів і програм входять у склад програмної документації й оформлюються відповідно до ДГСТ 19.701 – 90 (ICO 5807 – 85) "Схеми алгоритмів, програм, даних і систем". Водночас використовуються умовні графічні позначення (УГП), які вписуються в прямокутник. Сторони прямокутника мають такі розміри: a = 10, 15, 20 … через 5 мм, b = 1,5a або b = 2a.

Подання алгоритму у вигляді блок-схеми, яка складається з послідовності геометричних фігур (блоків, символів), кожен з яких відтворює зміст чергового кроку алгоритму, називається

графічним описом алгоритму. Усередині фігур коротко записують дію, яка виконується в цьому блоці.

Блок-схемою називається графічне зображення логічної структури алгоритму, в якому кожний етап процесу оброблення даних подається у вигляді блоків, що мають визначену конфігурацію залежно від виконуваних операцій. Графічні символи, їх розміри, а також правила побудови блок-схем визначені державними стандартами.

Основою цього способу є поняття символу дії, який зображається окремою геометричною фігурою, функціональним блоком (табл. 1.1).

Таблиця 1.1

Функціональні блоки

	Блок початку/завершення алгоритму
	Блок введення/виведення даних
	Арифметичний блок, модифікування (виконання операції або групи операцій, в результаті яких змінюються значення, форма подання або розміщення даних)
ні так	Умовний блок, розгалуження (вибір напрямку галуження алгоритму залежно від результату аналізу умови)
	Блок виклику підпрограми (функції, процедури)

Продовження таблиці 1.1	
	Блок циклу з параметром
	Міжсторінковий з'єднувач (вказівка зв'язку між роз'єднаними частинами схем алгоритмів та програм, розташованих на різних аркушах)
	Коментар (зв'язок між елементами схеми і поясненнями)
	З'єднувач (вказівка зв'язку між перерваними лініями потоку в межах однієї сторінки)

Під час створення блок-схеми алгоритму блоки із записаними в них командами з'єднуються між собою стрілками для визначення черговості виконання дій алгоритму.

Для запису команд всередині блоків використовується природна мова з елементами математичної символіки. Розробити алгоритм – це розбити задачу на етапи (більш прості задачі), які послідовно виконуються. Водночас чітко зазначається і зміст кожного етапу, і порядок їх виконання.

1.5. Основні керуючі структури алгоритмів

Алгоритми як процеси перетворення інформації мають певну класифікацію, яка відтворює особливості їх реалізування (хоча треба розуміти, що довільний підхід до спроби класифікування

має умовний характер). Загалом існує три типи алгоритмів: лінійний, розгалужений, циклічний.

Це три типові (базові) керуючі структури, які можна використовувати для розроблення алгоритмів різного ступеня складності:

– *лінійним алгоритмом* називається такий алгоритм, в якому не порушується природний порядок обчислень, тобто, лінійні команди виконуються послідовно одна за одною (подання цієї конструкції у блок-схемах здійснюється послідовністю блоків модифікування);

– *розгалуженим алгоритмом* називається алгоритм, який містить хоча б одну умову, в результаті аналізу якої здійснюється перехід до одного з можливих подальших кроків. Процес розгалуження організовується за допомогою логічного блоку;

– *циклічний алгоритм* містить повторення певну кількість разів з новими початковими даними певної послідовності команд, яка утворює тіло циклу. У циклі повинна існувати змінна (параметр циклу), яка при кожному наступному виконанні тіла циклу змінює своє значення і визначає кількість повторень. У простому циклі змінюється один параметр від заданого початкового до кінцевого значення з постійним кроком (залежно від вимог задачі параметр циклу може збільшуватися або зменшуватися). При кожному значенні параметра виконуються дії, які знаходяться всередині циклу, тобто тіло циклу. Вихід з циклу здійснюється після досягнення параметром циклу свого кінцевого значення.

Розглянемо деякі типові прийоми алгоритмізації обчислювальних процесів, які на практиці застосовуються або у вигляді окремих алгоритмів, або входять до складу більш складних алгоритмів. Водночас потрібно пам’ятати, що та ж сама задача може бути розв’язана різними способами.

Основними методами побудови алгоритмів є розроблення розгалужених алгоритмів, організація простих і вкладених циклів, обчислення суми, добутку, кількості, впорядкування одновимірних та багатовимірних масивів: обчислення максимального та міні-

мального значень масивів, сортування масивів за зростанням/ спаданням числових значень елементів тощо. Реальні задачі мають суперпозицію або композицію цих типових засобів. Практично довільний алгоритм містить один або декілька циклів, тобто ділянок, які виконуються багаторазово.

1.6. Лінійні алгоритми

Розглянемо лінійну керуючу структуру, яка означає, що дві або більше дій потрібно виконати послідовно одна за одною. У кожному блоці послідовності може бути проста дія або сукупність складних обчислень та перетворень.

Для подання такого алгоритму використовується алгоритмічна конструкція послідовного виконання, яка передбачає послідовне виконання дій не порушуючи природний порядок їх виконання.

Варто зауважити, що обчислення виразів є доволі складним процесом, який потребує концентрування уваги і часу, збереження проміжних результатів. Порядок обчислень у кожній мові визначається за пріоритетом операцій та дужками, використаними у записі виразу. Ці вирази можуть бути доволі складними, тому доцільно розбивати складний вираз на декілька більш простих, пам'ятаючи, що запис виразу довільної складності повинен бути лінійним.

Наприклад, задача обміну місцями числових значень двох змінних. Цей алгоритм застосовується дуже часто. Найпростіший спосіб обміняти місцями значення двох змінних a та b – використати третю змінну c. Під змінною розуміємо деяке місце в оперативній пам'яті, де можуть зберігатися довільні значення. Абстрактно це можна уявити як контейнер, який може бути заповнений рідиною. Коли необхідно обміняти вміст двох контейнерів, знадобиться третій.

Переливаємо: з a тимчасово у c; з b в a; c назад в b. Блок-схему алгоритму розв'язання задачі подано на рисунку 1.1.

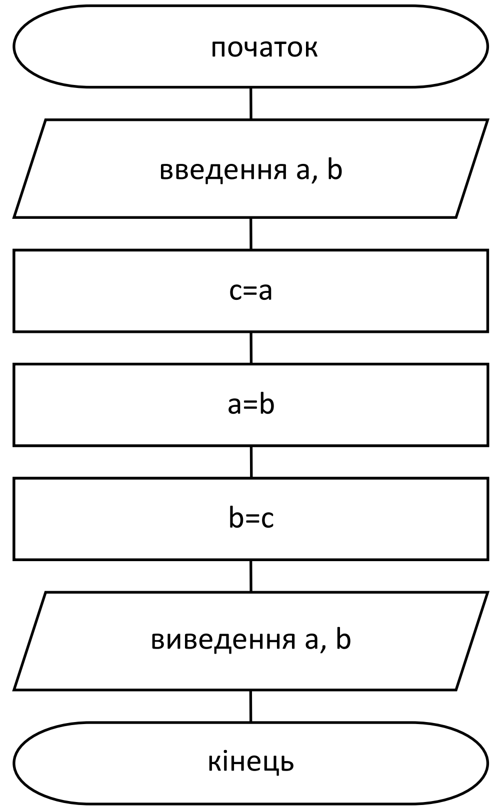


Рис. 1.1. Блок-схема алгоритму розв’язання задачі

1.7. Розгалужені алгоритми

У випадках, коли перетворення інформації може здійснюватися за різними схемами, залежно від властивостей початкових даних або проміжних результатів, використовуються *розгалужені алгоритми*. У таких алгоритмах передбачаються всі можливі варіанти оброблення інформації, кожний з яких розробляється як окрема гілка алгоритму, а вибір однієї з них для виконання здійснюється за результатом аналізу деякої умови, що відтво-

рює властивості інформації, яка використовується у процесі перетворення.

Для подання таких алгоритмів використовуються алгоритмічні конструкції розгалуження. Ця алгоритмічна структура залежно від результату аналізу певної умови здійснює вибір одного з альтернативних шляхів виконання алгоритму. Кожен із шляхів веде до спільного виходу, тому робота алгоритму триватиме незалежно від того, який шлях буде обраний.

Ця алгоритмічна структура передбачає виконання дій як у разі виконання, так і у разі невиконання заданої умови. Графічне подання такої структури у блок-схемах має такий вигляд, як на рисунку 1.2.

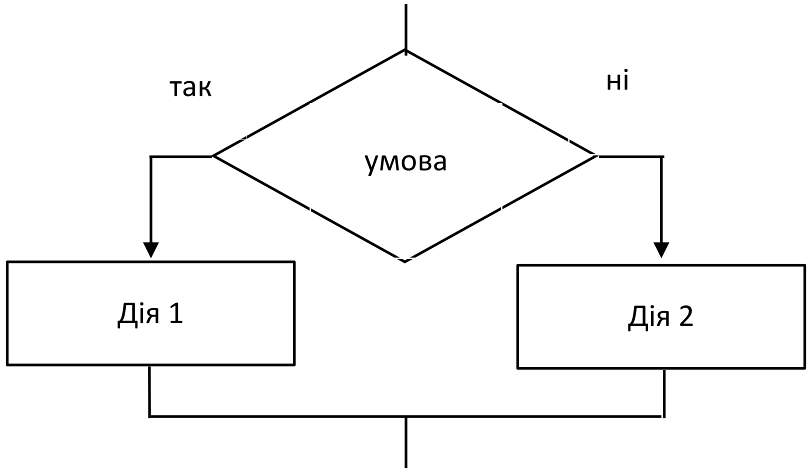


Рис. 1.2. Графічне подання розгалуженої структури у блок-схемах

Наприклад, побудувати блок-схему алгоритму з перевірки введеного числа на те, чи воно додатне, чи від'ємне. Блок-схему алгоритму розв'язання задачі подано на рисунку 1.3.

Побудувати блок-схему алгоритму задачі порівняння двох чисел a і b. Блок-схему алгоритму розв'язання задачі подано на рисунку 1.4.

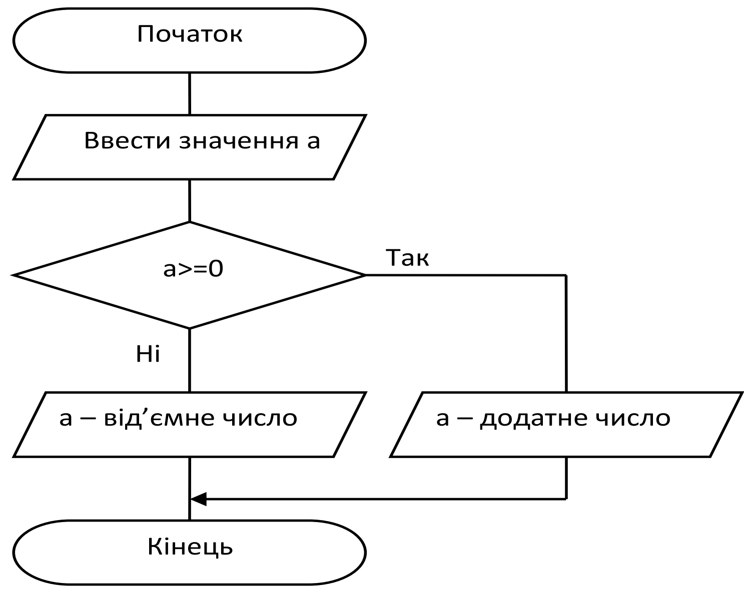


Рис. 1.3. Блок-схема алгоритму розв’язання задачі

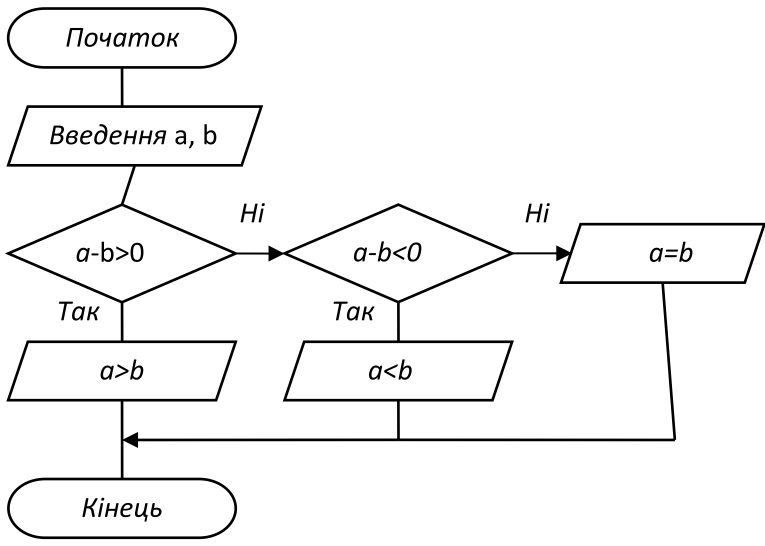


Рис. 1.4. Блок-схема алгоритму розв’язання задачі

Основою для розв’язання цієї задачі є обчислення різниці між a та b з подальшим аналізом цієї різниці. Якщо $a - b > 0$, тоді числове значення a буде більшим від b, якщо $a - b < 0$, тоді числове значення b буде більшим від a $b > a$.

Нарешті, якщо $a - b = 0$, тоді числові значення a і b дорівнюють одне одному.

1.8. Алгоритми циклічної структури

Алгоритмам циклічної структури, власне, притаманне повторення однотипних обчислень певну кількість разів.

Послідовність дій, які повторюються певну кількість разів, утворюють тіло (область дії) циклу.

Кількість повторень може бути відомою або завершення циклу відбувається за результатом аналізу якоїсь умови. Якщо число повторень циклу відоме, тоді такі цикли називають арифметичними. Якщо кількість повторень циклу наперед невідоме, а визначається тільки в процесі функціонування алгоритму, тоді такі цикли називають ітераційними. Кількість повторень у циклі повинна бути скінченною.

Кількість ітерацій циклу залежить від виконання або невиконання деякої умови, яка перевіряється на кожній ітерації циклу.

Розрізняють цикл з передумовою – коли спочатку аналізується умова, а потім, за результатом аналізу умови, виконується тіло циклу (деяка послідовність дій).

Цикл з постумовою – спочатку виконується, принаймні один раз, тіло циклу, а потім аналізується умова. За результатом аналізу умови або виконується наступна ітерація циклу, або здійснюється вихід з циклу.

Блок-схеми обидвох типів циклів подані на рисунку 1.5.

Для організації циклу з відомим числом повторень використовується блок циклу за параметром. У цьому блоці об’єднано 3 дії: ініціалізування початкового значення параметра циклу; його модифікування у процесі виконання циклу; аналіз умови закінчення (продовження) циклу.

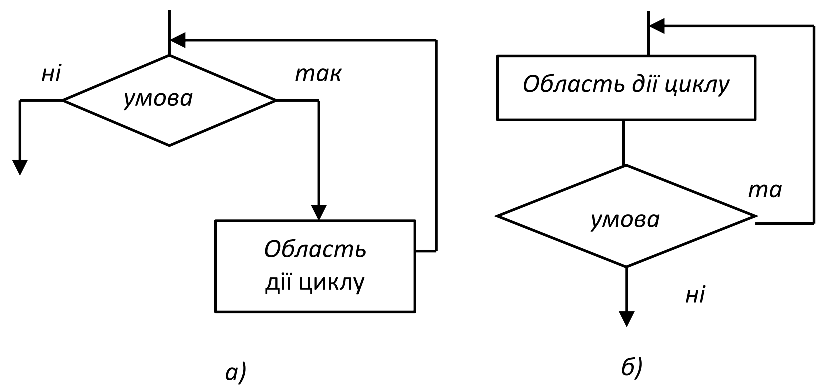


Рис. 1.5. Блок-схеми циклічних алгоритмів:
а) з передумовою; б) з постумовою

Структура циклу з відомим числом повторень подана на рисунку 1.6.

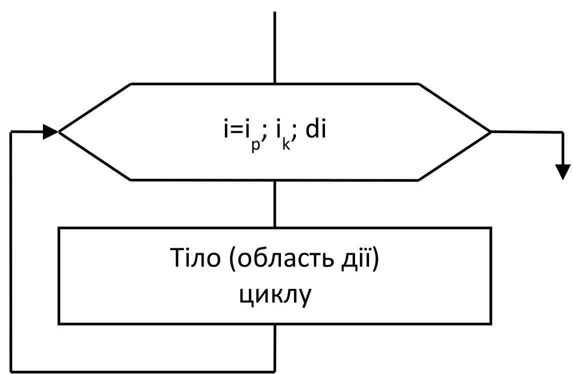


Рис. 1.6. Структура циклу з відомим числом повторень:
i_p – початкове значення параметру, i_k – кінцеве значення параметру,
di – крок модифікування параметра циклу

На першій ітерації циклу параметрові і присвоюється значення i_p, а по цьому виконується тіло циклу і збільшення значення

параметру на di. Тепер виконується аналіз умови $i <= i_k$. Керування циклом здійснюється на підставі порівняння поточного значення параметра циклу з його заданим кінцевим. Якщо поточне значення параметра циклу є меншим або дорівнює кінцевому, тоді виконується наступна ітерація, в іншому разі виконання циклу завершується (здійснюється вихід з циклу).

Якщо до початку циклу $i_p > i_k$, тоді область дії циклу не виконується жодного разу. Після нормального завершення циклу параметр (керуюча змінна) має невизначене значення і тому застосовувати його у подальшому, як лічильник кількості виконаних ітерацій циклу неможливо.

1.9. Приклади проєктування алгоритмів розв'язання класичних типів задач

Приклад 1. Розглянемо задачу табулювання функції однієї незалежної змінної, яка полягає у обчисленні таблиці значень функції Y = a + sin(bx) за відомими значеннями коефіцієнтів a та b і аргументу x, що змінюється від -4 до 6 з кроком 2. Початкове значення аргументу x позначимо як xp, кінцеве – xk і крок зміни поточного значення аргументу x як dx. Тобто, у цьому прикладі проста змінна x є аргументом функції Y і змінюється з кроком dx. На рисунку 1.7 подано блок-схему алгоритму розв'язання поставленої задачі, яка охоплює:

– блок введення значень коефіцієнтів a, b та кроку зміни аргументу – dx;

– блоки присвоювання початкового значеня змінній x = xp;

– обчислення значення функції Y з наступним виведенням значень Y і x;

– поточне значення x збільшується на крок dx;

– логічний блок, який керує циклом, містить аналіз умови x <= xk і, якщо x перевищить значення xk, забезпечує вихід з циклу. Якщо у результаті аналізу умова виконується (поточне значення x є меншим або дорівнює xk), тоді повертаємося до обчислення значення функції Y.

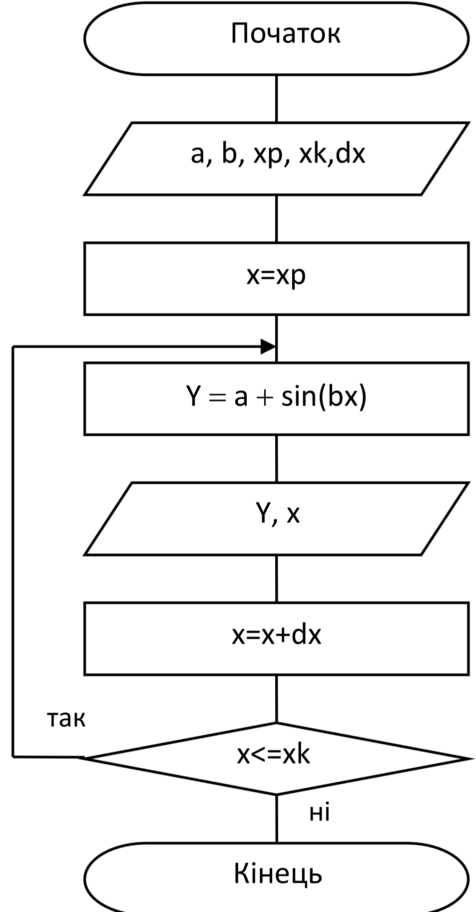


Рис. 1.7. Блок-схема алгоритму розв'язання задачі

Це приклад формування циклу з постумовою, тому що аналіз умови виконується після виконання тіла циклу.

Приклад 2. Подамо трохи інший підхід до розв'язання цієї ж задачі, якщо кількість повторень циклу буде відомою. Число повторень циклу N обчислюється за формулою:

$$N = ((xk - xp)/dx) + 1.$$

Поточне значення аргументу x буде обчислюватися за рекурентною формулою, у якій i є поточним значенням параметра циклу на кожній ітерації:

$$x = xp + (i - 1)*dx.$$

Тепер блок-схема алгоритму розв'язання задачі буде мати вигляд як на рисунку 1.8.

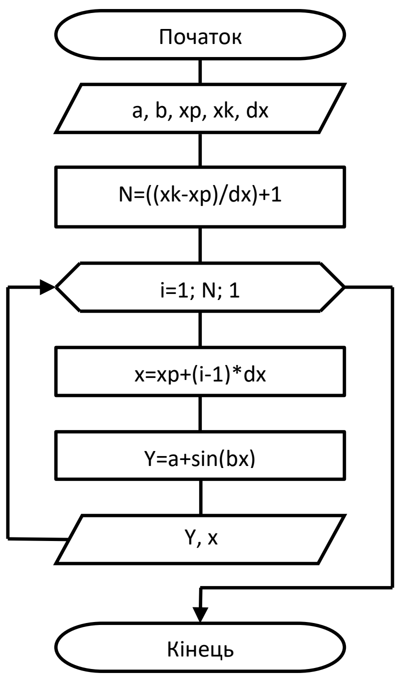


Рис. 1.8. Блок-схема розв'язання задачі

Приклад 3. Накопичення суми елементів масиву a, який складається з n елементів. Для розроблення алгоритму розв'язання

цієї задачі необхідно ввести значення елементів масиву a та розмір масиву n. Символом S позначимо суму елементів масиву.

Змінній S перед оргацізацією циклу за параметром i присвоюємо початкове значення нуль. У тілі циклу на кожній ітерації до попереднього значення S будемо додавати значення поточного елемента a_i.

Блок-схема розв'язання задачі подана на рисунку 1.9.

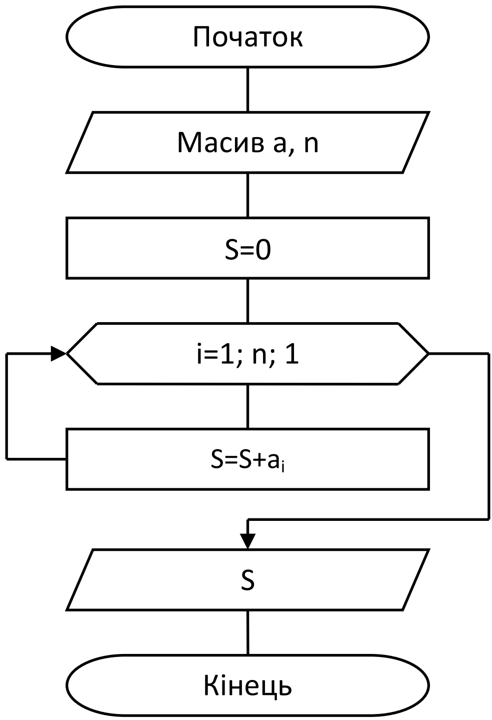


Рис. 1.9. Блок-схема алгоритму розв'язання задачі (приклад 3)

Приклад 4. Обчислити кількість додатних елементів масиву a розміром n.

Здійснюємо введення початкових даних, тобто введення значень елементів масиву a та розмір масиву n.

Символом k позначено кількість додатних елементів масиву a. Перед початком циклу за параметром i присвоюємо k початкове значення k=0. У тілі (області дії) циклу на кожній ітерації перевіряємо значення поточного елемента масиву a_i. Якщо поточне значення a_i є додатним ($a_i > 0$), тоді значення k збільшуємо на 1 (k = k + 1).

За свою суттю ця задача схожа до попередньої, накопичення суми елементів масиву. Різниця полягає у тому, що накопичення суми "одиничок", а не суми значень елементів масиву здійснюється за результатом аналізу поставленої умови, коли значення поточного елемента масиву є більшим від нуля ($a_i > 0$).

Блок-схема алгоритму розв'язання задачі обчислення кількості додатних елементів масиву a розміром n подано на рисунку 1.10.

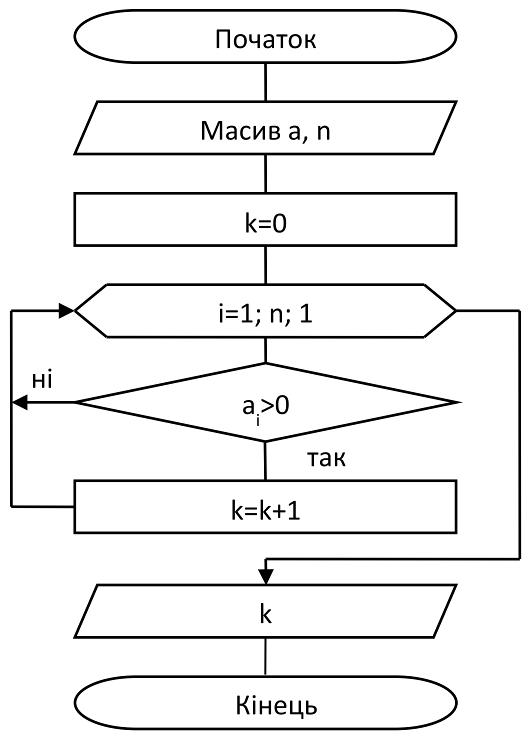


Рис. 1.10. Блок-схема алгоритму розв'язання задачі (приклад 4)

Приклад 5. Заданий масив чисел X, який складається з n елементів. Обчислити максимальний елемент і його порядковий номер у масиві. Блок-схема розв'язання задачі подана на рисунку 1.11.

Позначення: max – максимальний елемент масиву; k – порядковий номер максимального елемента в масиві.

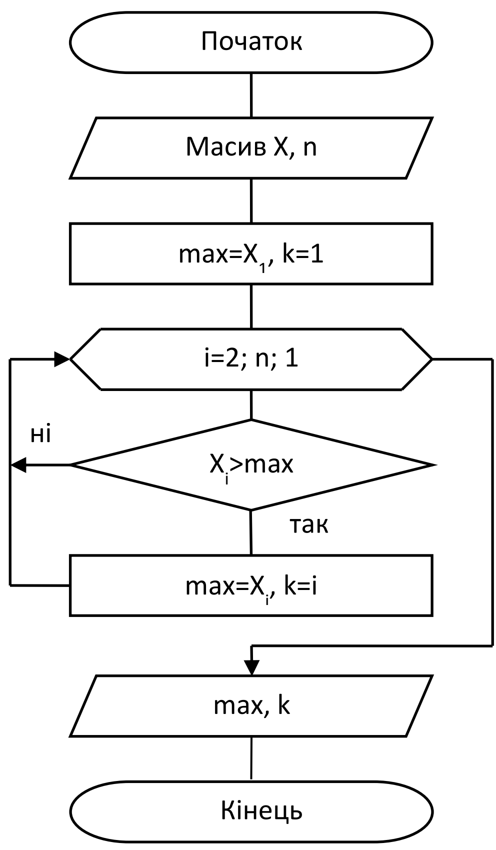


Рис. 1.11. Блок-схема алгоритму розв'язання задачі (приклад 5)

Розв'зання задачі полягає у такому:

– здійснюємо введення початкових даних (елементів масиву X та його розміру n);

– max присвоюємо значення X_1, а $k = 1$ значення індексу першого елемента (вважаємо, що перший елемент масиву є максимальним);

– організовуємо цикл за параметром i, який буде змінювати свої значення від 2 до n з кроком 1 (порівняння поточного елемента X_i з max);

– якщо значення поточного елемента масиву X_i виявиться більшим від max, тоді максимальному присвоюємо значення цього елемента, а k – присвоюємо значення i;

– останній, більший від max і буде максимальним елементом масиву, а його індекс порядковим номером у масиві X.

Приклад 6. Задана матриця B_{ij} ($i = 1…n$, $j = 1…m$), $n = 4$, $m = 5$. Необхідно для кожного рядка матриці обчислити кількість від'ємних елементів та записати їх в одновимірний масив. Блок-схема розв'язання задачі подана на рисунку 1.12.

Головна умова правильного розв'язання задач із впорядкування елементів матриць полягає у розумінні порядку змінювання індексів її елементів. Перший індекс (i) завжди визначає номер рядка, другий (j) – номер стовпця.

Алгоритм розв'язання поставленої задачі виконується так:

– спочатку послідовно вводяться значення елементів масиву B_{ij} ($i = 1…n$, $j = 1…m$), тобто блок введення передбачає використання двох циклів – "рядками" (за параметром i) та "стовпцями" (за параметром j). *Двовимірні масиви здебільшого вводяться "рядками"*, тобто цикл за параметром i є зовнішнім, а цикл за параметром j – внутрішнім;

– з огляду на те, що обчислюється кількість від'ємних елементів для кожного рядка матриці, тому зовнішній цикл передбачено за параметром i;

– накопичення кількості від'ємних елементів буде записуватися у змінну k (її початкове значення перед організацією циклу за параметром j буде дорівнювати нулю, тобто задаючи черговий номер рядка значенням параметра циклу за i обнулюється значення кількості від'ємних елементів у цьому рядку, $k = 0$);

– для здійснення процесу накопичення кількості від'ємних елементів рядка потрібно перебирати всі його елементи, тобто

використати цикл за параметром j, який буде внутрішнім відносно циклу за параметром i;

– для організації лічильника k необхідно скористатися типовим прийомом алгоритмізації, тобто k = k + 1;

– для запам’ятовування результатів аналізу значень елементів кожного рядка (CV_i = k) необхідно скористатись типовим прийомом алгоритмізації – формуванням робочого масиву CV_i (i = 0...n), виведення елементів якого здійснюється після перегляду всіх рядків матриці.

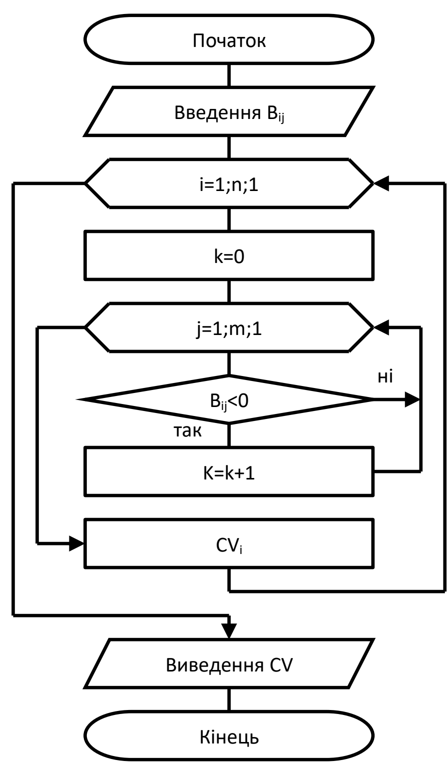


Рис. 1.12. Блок-схема алгоритму розв’язання задачі (приклад 6)

У процесі організації вкладених циклів рекомендується дотримуватися такого правила: *параметр циклу, який визначений останнім, повинний мінятися першим* (так, параметр j визначається після i, тому насамперед змінюється параметр j).

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

Зауважимо, що у межах посібника блок-схеми алгоритмів подано без урахуванням важливої особливості мови C++: нумерування елементів масивів починається з нуля (0), а не з одиниці (1).

Розділ 2

ПАРАДИГМИ ПРОГРАМУВАННЯ

Слово парадигма походить з грецької та означає стиль міркування, спосіб дій або набір концепцій у певній галузі знань. Останніми десятиліттями, після того, як американський вчений Роберт Флойд під час вручення йому Тьюрінгової медалі ACM 1978 року виступив з лекцією "Парадигми програмування", термін "парадигми" став часто вживаним у програмуванні, а тому варто його розглянути детальніше.

2.1. Парадигми і мови програмування

Поняття програми залежить від того, на якого виконавця вона розрахована. Так, речення "обчисліть корені квадратного рівняння $x^2+2x-3=0$" є прикладом програми, виконавцем якої може бути учень або студент, причому різні виконавці можуть робити це по-різному: один – користуючись теоремою Вієта, інший – за допомогою дискримінанту. Це приклад декларативної (описової) парадигми: програма відповідає на запитання "що?" – що шукати? що будувати? що рахувати? Водночас вибір відповіді на запитання "як? " залишається за виконавцем.

Інша програма могла б виглядати так: "Додайте до одиниці три, обчисліть квадратний корінь з отриманої суми, відніміть його від мінус одиниці. Це буде перший корінь. Потім додайте його до мінус одиниці. Це буде другий корінь". Характерну особливість цієї програми можна визначити як імперативність: "спочатку роби одне, а потім інше". Вона є прикладом імперативної (наказової) парадигми програмування.

Зазвичай декларативна парадигма потребує досконалішого виконавця, тоді як застосування імперативної парадигми покладає більше відповідальності на програміста, адже він повинен дати більш детальний рецепт розв'язування задачі. Декларативне програмування – це щось на зразок замовлення на кухні ресторану з доставленням, де буде приготовано за Вас обід, тоді як імперативне потребує вправності та майстерності Попелюшки у виконанні забаганок вередливої мачухи.

Поділ на "що? " і "як?" у програмуванні відтворює також природний перебіг процесу розв'язування задач. Спочатку визначаємося з тим, що треба зробити, проєктуємо інтерфейси – виникає щось на зразок декларативної програми. Потім з'ясовуємо наявність виконавця, здатного цю програму зрозуміти і, можливо, виконати. Якщо його підібрано, оцінюємо характеристики виконання ним програми та порівнюємо їх на відповідність заданим вимогам. Якщо придатного виконавця немає, беремо найбільш підхожого з наявних і підсилюємо його можливості додатковими імперативними засобами, необхідними для розв'язання поставленого завдання.

Зауважимо, що сам професор Джон Бекус – винахідник першої мови наказового програмування *Fortran* – успішно працював над пошуком нових парадигм, зокрема декларативної парадигми функціонального програмування, що також викладено у його Тьюрінговій лекції 1977 року. Одночасне застосування кількох парадигм, або мультипарадигменність, дає можливість подивитися на задачу з різних поглядів, завдяки чому поєднуються різні підходи і збільшуються шанси на відшукання кращого розв'язку.

Наголосимо, що *парадигми програмування* – це моделі, які відтворюють спосіб мислення розробника програми. Мова програмування може підтримувати або не підтримувати ту чи іншу парадигму. У першому випадку застосування парадигми стає зручним, тобто простим, безпечним і ефективним, в іншому – складним і ненадійним. Ми розглянемо три основних наказових парадигми – процедурне, об'єктне (модульне) і об'єктно-орієнтоване (ієрархічне) програмування (рис. 2.1) – та одну допоміжну – узагальнене програмування.

Імперативне програмування

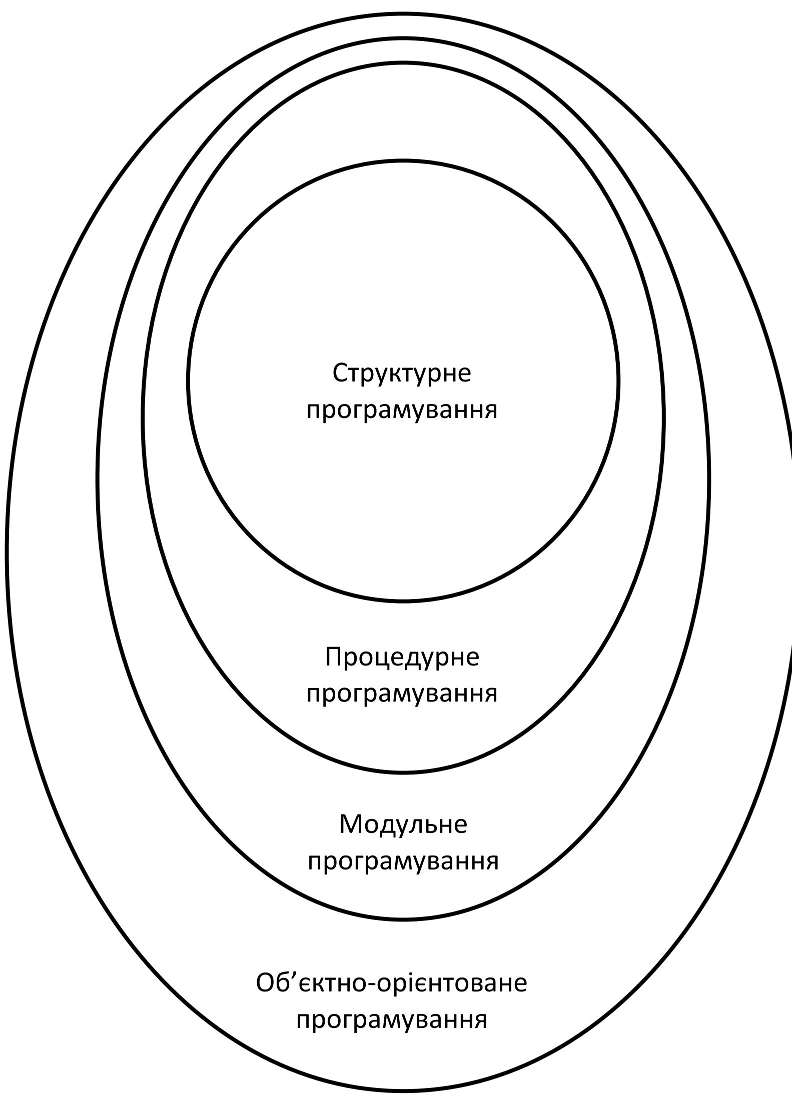


Рис. 2.1. Парадигми імперативного програмування

П’ятдесят років розвитку програмування стали роками пошуків та утвердження виражальних засобів, роками зміни поколінь комп’ютерів, програмного забезпечення і мов програмування. На рисунку 2.2 подано мови програмування, які зробили вагомий внесок у розвиток сучасних парадигм. Так, мова програмування *Fortran* (*FORmule TRANslator* – перекладач формул) уперше вирішила проблему автоматизації програмування математичних формул, а її компілятори уперше впоралися із задачею роздільної компіляції.

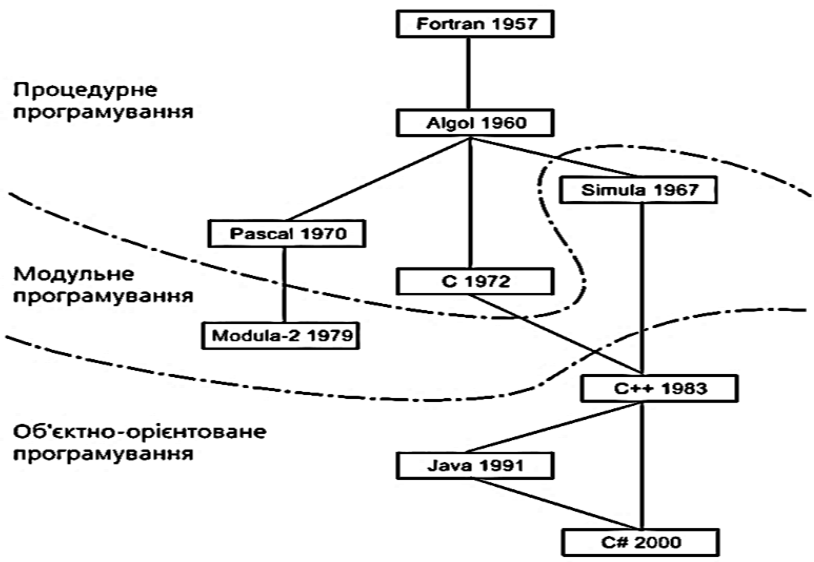


Рис. 2.2. Хронологія мов і парадигм програмування

Мова програмування *Algol* (*ALGOrithmic Language* – алгоритмічна мова), у якій було запроваджено блокову структуру програми, стала першою мовою структурованого програмування із сучасними розгалуженнями і циклами. Майже одночасно з’явилися дві інші знакові мови програмування – *Pascal* і С, які дотримувалися майже діаметрально протилежних підходів. *Pascal* – прямий нащадок *Algol* – доповнив його структури керування розвиненими структурами даних. Він стандартизував модель обчислювальної

Pascal – машини, призначеної для виконання програм цією мовою, і усі реальні обчислювальні середовища повинні були налаштовуватися на стандартну модель. Мова програмування С не передбачала стандарту обчислювача, а навпаки, містила засоби налаштування програми на конкретне обчислювальне середовище.

Проблема складності програмування на основі процедурної парадигми стимулювала пошук нових підходів. Так з'явилася концепція абстрактних типів даних як способу об'єднання даних і засобів їхнього оброблення та одночасно як методу відокремлення специфікацій (інтерфейсів) від їх реалізування. Типи даних *Pascal* розвинуті в абстрактні типи даних у вигляді модулів мови *Modula*-2.

Особливо важливий крок зроблено внаслідок залучення до мови С засобів мови моделювання *Simula* (*SIMUlation Language* – мова моделювання), в якій свого часу вперше з'явилися класи і об'єкти. З цього союзу виникла мова програмування С++, яка сама пройшла еволюційний шлях вдосконалень і розвитку.

Зазначимо, що хоча мова *Modula*-2 не набула широкого розповсюдження, закладені у неї ідеї призвели до розширення мови *Pascal* класами та об'єктами. Так, під назвою *Object Pascal* виникла об'єктна версія мови *Pascal*, реалізована в системі програмування *Delphi*. Зрештою, назву *Delphi* одержала й сама мова програмування, яка продовжила давню традицію конкуренції між *Pascal* і С.

Головне досягнення С++ полягає у зміні парадигми програмування (*paradigm shift*) з процедурної на об'єктно-орієнтовану, яка визначає стандарт розроблення програмного забезпечення. Головним доробком об'єктно-орієнтованої парадигми стала ієрархічність програмних структур, яка відтворюється в агрегації об'єктів та успадкуванні класів.

Продуктивність цього підходу полягає у його природності та гнучкості. Справді, програма, подана у вигляді архітектури класів та об'єктів, більш повно, ніж набір функцій і процедур, відповідає реальним об'єктам, обчислювальною моделлю яких вона є. Внаслідок цього з'явилася можливість проєктувати програми більш звичними методами інженерного проєктування. Водночас властивість ієрархічності дала змогу об'єднувати програмні структури у складні архітектурні конструкції, максимально використовуючи наявний код, налагодження якого тепер не потребує перепрограмування.

2.2. Структурне програмування

Основою структурного програмування є поєднання теорії програмування та особистого досвіду висококваліфікованих програмістів з урахуванням сучасних вимог до програм та промислового характеру їх створення. Головною вимогою, якій повинна відповідати програма – працювати відповідно до специфікації та адекватно реагувати на довільні дії користувача. Крім того, розроблення програми повинно завершитися у встановлені терміни і надавати можливість вносити необхідні зміни та доповнення.

Структурне програмування – це технологія створення програм, яка дозволяє шляхом дотримання певних правил зменшити час розроблення і кількість помилок, а також полегшити модифікування програми.

Структурний підхід охоплює усі етапи проєкту – специфікацію, проєктування, тестування.

Структурний підхід до програмування надав змогу успішно створювати достатньо великі проєкти, але складність програмного забезпечення продовжувала збільшуватись і вимагалися щораз нові засоби для вирішення цієї ситуації. Ідеї структурного програмування набули подальшого розвитку в об’єктно-орієнтованому програмуванні (ООП). ООП – це технологія, яка дозволяє досягнути простоти структури і керованості дуже великих програмних систем.

2.2.1. Структурний підхід до проєктування програм

За часів стихійного програмування хорошими програмістами вважалися ті, хто створював достатньо хитромудрі програми, які займали мінімальний обсяг пам’яті та швидко виконувалися. Це було цілком природно, враховуючи тодішні можливості обчислювальної техніки. Як результат такого програмування створено програми, які було важко (якщо взагалі можливо) зрозуміти іншим фахівцям. Інколи навіть автори таких програм з часом важко розуміли власне творіння. Внесення необхідних змін у таку програму

робило ситуацію ще більш заплутаною, ніж вона була до цього. Подібні проєкти отримали назву BS-програм – абревіатура від "*bowl of spaghetti*" – блюдо спагеті, бо саме так виглядала готова програма під час спроби відтворити всі переходи між її настановами.

Засновник структурного програмування Е. Дейкстра навіть проголосив, що "кваліфікація програміста обернено пропорційна кількості настанов безумовного переходу в його програмах". Структурне програмування ще тоді називали "програмуванням без `goto`", хоча для багатьох це був екстремальний погляд. Насправді йдеться про те, щоб не використовувати настанови переходу без особливої потреби. Насамперед структурне програмування мало своєю метою позбавитись від поганої структури самої програми. Ще однією метою було створення таких програм, які були б легко зрозумілими навіть без їх авторів, адже "програми пишуться для людей – комп'ютером вони лише обробляються". Зміст цієї фрази полягає у тому, що транслювання та виконання програми довільної структури за допомогою комп'ютера дійсно не викликає ніяких труднощів. А от трудомісткий процес перевірки правильності роботи програми, внесення відповідних виправлень і логічних змін доводиться виконувати людині.

Отже, *структурне програмування* – це методологія й технологія розроблення поважних програмних проєктів, яка об'єднує способи розроблення структури програми, зручної для читання та розуміння людиною, стеження за логікою її роботи, внесення до неї виправлень та інших змін. Згідно з думкою Н. Вірта, "структуризування є принциповим інструментом, який допомагає програмісту систематично аналізувати і синтезувати складні програми, зберігаючи про них повне уявлення". Ідеї структурного програмування з'явилися на початку 1970-х роках у компанії *IBM*, у її розробленні брали участь відомі вчені Е. Дейкстра, Х. Милі, Є. Батіг, С. Хоор.

Структурне програмування ґрунтується на таких основних принципах:

– програмування має здійснюватися зверху-униз;

– вся програма розв'язання складної задачі має бути поділена на модулі з одним входом і одним виходом (оптимальний розмір модуля – кількість рядків, які поміщається на екрані дисплея);

– логіка алгоритму й програми має допускати тільки три основні структури: послідовне виконання, розгалуження і повторення. Недопустиме передавання керування у довільну точку програми;

– при розробленні коду програми супровідна документація має створюватися одночасно із програмуванням, у вигляді коментарів до неї.

З погляду структурного програмування, правильна програма – це програма, структура якої містить тільки основні елементи, і жоден з них не є недоступним і не допускає так зване зациклювання. Правильна програма має тільки один вхід і тільки один вихід. У правильній програмі не має бути таких частин, які ніколи не виконуються.

2.2.2. Принцип поділу програми на окремі модулі

Принцип поділу програми на окремі модулі (у колах програмістів його називають принципом модульності програми) полягає у тому, що довільну складну програму доцільно поділити на логічно незалежні частини (модулі), дотримуючись водночас певних зв'язків між ними. Історично поняття модульної програми виникло раніше, ніж були сформульовані принципи структурного програмування, проте ця ідея виявилася просто необхідною складовою нової технології програмування разом з аналітичним проєктуванням.

Модуль – це послідовність логічно пов'язаних команд, який оформлено у вигляді окремої програми. Ці допоміжні програми можна розробляти й аналізувати окремо та незалежно одну від іншої, використовуючи їх потім у основній програмі або інших допоміжних програмах. Структурний підхід до розроблення програм і принцип її модульності також призвів до ідеї розподілу робіт з-поміж розробників програм.

Поняття модуля цілком логічно з'явилося на відповідному етапі аналітичного програмування: модуль – частина програми, яка

розв'язує порівняно нескладну задачу, логічно незалежну від інших задач. Здебільшого, модулі – це підпрограми (процедури або функції), які мають певні властивості, а саме:

– єдиний вхід, єдиний вихід (деякі мови дають змогу існування декількох входів або виходів);
– окреме компілювання кожного модуля;
– кожний модуль доступний за своїм ідентифікатором;
– модуль може викликати інший модуль;
– модуль не має зберігати історію своїх викликів (інакше може виникати так званий побічний ефект);
– модуль має бути невеликим порівняно зі всією програмою;
– кожен модуль відповідає за виконання тільки однієї задачі;
– незалежність функціонування (заміна модуля на аналогічний не впливає на роботу всієї програми).

З часом, коли принцип модульності став підтримуватись різними мовами програмування, на перший план висунулась вимога логічної та програмної незалежності модулів. Необхідно зауважити, що повної незалежності між модулями бути не може. Залежність між модулями існує тоді, коли:

– використовуються спільні списки параметрів;
– вони користуються спільними (глобальними) змінними;
– модулі програми залежать від структури даних цієї програми;
– модулі програми залежать від логіки функціонування програми (від того чинника, що модуль може викликатися іншими модулями).

Отже, модулі мають бути незалежні в межах інтерфейсу програми і структури даних. Практика програмування засвідчила, що чим вищий ступінь незалежності модулів, тим простіше розібратись в окремих модулях і в самій програмі загалом; тим менша ймовірність появи нових помилок під час усунення старих, або внесення змін у програму, тобто менша ймовірність так званого хвильового ефекту. Із сказаного випливає, що не варто без крайньої на те потреби використовувати в модулях глобальні змінні. Всі зв'язки між модулями мають підтримуватися через списки параметрів.

2.2.3. Методологія покрокового деталізування програми

У процесі створення програми розв'язання складної задачі застосовується методологія її покрокового деталізування ("проєктування зверху вниз"). Водночас складна задача ділиться на декілька простих, які можуть ділитися на серію ще простіших тощо.

Процес покрокового деталізування вважається завершеним, коли задачі поточного рівня стають достатньо простими для незалежного їх розв'язання. Потім результати програмування простих задач компонуються у загальний алгоритм.

Технологія структурного програмування часто вимагає розв'язання різних додаткових задач. Програму для виконання основної задачі назвемо основною програмою. Тоді програма для розв'язання допоміжної задачі, виокремленої в окрему структуру, називатиметься допоміжною програмою. Вона призначена для досягнення деякої допоміжної мети на шляху розв'язання основної задачі.

Одна і та сама допоміжна програма може бути використана і в основній, і в іншій допоміжній програмі. Кожна допоміжна програма має мати своє унікальне ім'я, за яким її ідентифікують з-поміж інших програм і за цим іменем вона викликається з довільної іншої програми.

Виконання допоміжної програми з потрібними початковими даними та/або передавання результатів розв'язання в основну програму здійснюється за допомогою параметрів програми. Параметри, які необхідно вказати перед початком роботи допоміжної програми, називаються аргументами програми.

Ці параметри дають змогу змінити вхідні дані перед початком роботи допоміжної програми. Параметри, значення яких визначаються внаслідок роботи програми, називаються результатами роботи програми.

Допоміжна програма може зовсім не мати параметрів або мати якийсь один тип параметрів. Параметри, які використовуються для опису початкових і результуючих даних програми, називаються

формальними. Під час конкретного виконання допоміжної програми формальні параметри замінюються на фактичні, тобто під час виклику допоміжної програми потрібно вказати її фактичні параметри.

Результатом виконання деякої допоміжної програми може бути одна або декілька результуючих величин, які є в переліку параметрів, або деяка дія (наприклад виведення повідомлення, відтворення звуку) без результуючої величини. Такі допоміжні програми називаються програмами-процедурами.

Програми-функції – це допоміжні програми, які виконують деяку послідовність різних дії, результатом виконання яких є один результат, що зазвичай присвоюється безпосередньо імені функції. Ім’я функції використовується в командах як ім’я звичайної змінної або деякої константи.

Програми-процедури і програми-функції забезпечують практичне реалізування методології структурного програмування. Виклик допоміжної програми має різний вигляд: виклик допоміжної програми-процедури здійснюється за допомогою спеціальної команди; виклик програми-функції відбувається безпосереднім вказанням імені функції з фактичними аргументами в деякому арифметичному або логічному виразі.

Виконання цих команд відбувається так:

– формальні аргументи допоміжної програми замінюються фактичними значеннями, вказаними в команді виклику програми у переліку фактичних параметрів;

– якщо в переліку фактичних аргументів є математичні або логічні вирази – тоді спочатку обчислюються вони;

– виконуються всі команди допоміжної програми з використанням фактичних аргументів;

– отримані результати присвоюються фактичним іменам результатів (іменам фактичних змінних, які використовуються як фактичні результати в програмах-процедурах або імені самої програми-функції).

Після виконання допоміжної програми виконується настанова основної програми, записана після настанови виклику допоміжної програми.

2.3. Процедурне програмування

Процедурне програмування зображає програму у вигляді набору алгоритмів, для оформлення яких можуть бути застосовані іменовані програмні блоки – процедури та функції. В останньому випадку передбачено наявність механізмів передавання параметрів і повернення результату.

Спочатку процедурне програмування користувалося довільними засобами керування, зокрема переходом за позначкою (`go to`) – однією із найуживаніших настанов керування в *Fortran*.

У 1968 році голландський вчений Едсгер Дейкстра вперше закцентував на проблемах, що виникають у програмах з неконтрольованими переходами, а в 1970 році проголосив новий напрям, який він назвав структурним програмуванням.

Структурне програмування (не зовсім вдалий переклад англійського *structured programming* – структурне програмування) – це варіант процедурного, який використовує лише три типи структур керування: послідовне виконання дій, розгалуження і цикл. Не дивно, що *Fortran* не підтримував цю парадигму: в наборі його засобів не було циклів за умовами. Починаючи з *Algol*, цикли стають основним засобом організації обчислень у програмі.

Професор Ніклаус Вірт, автор мови програмування *Pascal*, відібрав до неї лише прості в поясненні та легкі в реалізуванні конструкції. Внаслідок сильного типізування програми на *Pascal* вирізняються високою надійністю, мобільністю, їх легко читати і розуміти завдяки дисципліні програмування, яка продиктована вжитою парадигмою.

Але водночас застосування мови *Pascal* гальмувалося складністю виходу за межі віртуальної машини, потребою ефективного застосування наявної апаратури. Головним критерієм, застосованим Д. Річі до створеної ним мови С, стала саме гнучкість використання особливостей конкретної апаратури та ефективність виконання програм.

2.4. Об'єктне (модульне) програмування

Процедурна парадигма віддала належне алгоритмічній компоненті програмування. Але зі збільшенням обсягу програм і складності даних з'явилася нова проблема, а саме проблема структурної організації даних, найбільш влучно висловлена Віртівською формулою: "алгоритми + структури даних = програми".

Поняття модуля як абстракції даних вперше запропонував Девід Парнас у 1972 році. Головна ідея модульності даних полягає у забезпеченні доступу до них і оперування ними незалежно від способу їхнього конкретного кодування у пам'яті комп'ютера. Самі дані разом із процедурами їх оброблення вмонтовують (інкапсулюють) в окрему одиницю програми.

Модулі мають дві головні риси. По-перше, вони об'єднують структури даних з алгоритмами їхнього оброблення. По-друге, у них відокремлено специфікацію від реалізування інкапсульованих у модулі конструкцій, і це перетворює модуль на так званий абстрактний тип даних (*abstract data type*), на що вказав Джон Гуттаг.

2.5. Об'єктно-орієнтовне програмування

Об'єктно-орієнтована парадигма розвиває об'єктне (модульне) програмування засобами створення ієрархій об'єктів і класів. Об'єктно-орієнтоване програмування, за метафорою Б'єрна Страуструпа, – це "високоінтелектуальний синонім доброго програмування".

Дійсно, хоча нові парадигми програмування з'являються не так часто – приблизно одна на десятиліття, лише деякі з них, як, наприклад, структурне програмування, стають справжніми "довгожителями". Той факт, що об'єктно-орієнтовану парадигму успішно використовують упродовж більш ніж чверті сторіччя, сам собою є вагомим підтвердженням її життєспроможності.

Алгоритми, реалізовані у процедурному програмуванні, надто конкретні. Довільне модифікувння – це вже новий алгоритм, і тому кількість використовуваних процедур і функцій, як і витрати

на їхнє розроблення або перепрограмування, надмірно зростають. Модульне програмування групує алгоритми в модулі (класи), інкапсулюючи разом з алгоритмами і відповідні структури даних. Тепер залишається зробити наступний крок – побудувати ієрархічну структуру.

Для чого потрібні ієрархії? Чому саме ієрархії виявилися головною рисою новітніх технологій програмування? По-перше, ієрархічні структури дають змогу керувати складністю програмних проєктів, відділяючи внутрішню складність конструкції від її зовнішнього використання. По-друге, ієрархічні структури забезпечують ефективні будівельні блоки для конструювання програм, надаючи ємніші програмні конструкції, ніж окремі процедури або функції. І, нарешті, ієрархічні структури забезпечують можливість пристосування до нових умов вже наявних програмних кодів, не втручаючись у їхню внутрішню будову.

Ієрархії можуть бути двох типів. Перший – це бути частиною чогось. Наприклад, грань є частиною багатогранника, ребро – грані, вершина – ребра. Цей тип ієрархії дає змогу збирати об'єкти з частин, які самі є об'єктами. Інший, складніший тип дає можливість будувати узагальнення або, навпаки, конкретизації. Наприклад, овал і багатокутник є конкретизацією плоскої фігури, коло – овалу, чотирикутник – багатокутника; подальшими конкретизаціями чотирикутника можуть бути паралелограм, прямокутник, ромб, квадрат. Те, що квадрат, ромб або прямокутник є повноцінними паралелограмами, дає їм змогу користуватись усіма програмними засобами, створеними для паралелограмів; паралелограм, зі свого боку, є повноцінним чотирикутником і так далі. Цей принцип, відомий під назвою *reusable* – знову (або навіть заново) вживаний, – став одним із найважливіших досягнень об'єктно-орієнтованої парадигми. Щоб знову використати вже наявне програмне забезпечення у більш конкретизованих або навіть зовсім несподіваних умовах, достатньо дописати лише ту його частину, яка стосується особливостей відповідного конкретизування. Цей принцип отримав назву *programming by difference* або дописування програм.

І, нарешті, об'єктно-орієнтована парадигма доводить до логічної завершеності принцип моделювання реального світу, а точніше – тієї його частини, абстракцією якої є програма.

За такого підходу програма складається з об’єктів, які відповідають реальним поняттям або предметам, а її виконання зводиться до взаємодії цих об’єктів, що є абстракцією реальної взаємодії їхніх прототипів. Все це разом забезпечує об’єктно-орієнтованому підходу беззаперечне лідерство у галузі розроблення програм.

2.5.1. Основні ознаки об’єктно-орієнтованого програмування

На думку Алана Кея, якого вважають одним з "батьків-засновників" ООП, об’єктно-орієнтований підхід до розроблення сучасних програмних продуктів полягає в такому наборі основних принципів:

– все можна подати у вигляді об’єктів;

– всі дії та розрахунки виконуються шляхом взаємодії (обміну даними) між об’єктами, при якій один об’єкт потребує, щоб інший об’єкт виконав деяку дію. Об’єкти взаємодіють між собою, надсилаючи і отримуючи повідомлення. Повідомлення – запит на виконання дії, доповнений набором аргументів, які можуть знадобитися при її виконанні;

– кожен об’єкт має незалежну пам’ять, яка складається з інших об’єктів;

– кожен об’єкт є представником (примірником) класу, який відтворює загальні властивості об’єктів;

– у класі задається поведінка (функціональність) об’єкта, тобто усі об’єкти, які є примірниками одного класу, можуть виконувати одні й ті ж дії;

– класи можуть бути організовані у єдину деревоподібну структуру з спільними коренями, яка називається ієрархією успадкування;

– стан пам’яті та поведінка об’єкта, зв’язані з примірниками деякого класу, стають доступні довільному класу, розташованому нижче в ієрархічному дереві.

Отже, програма складається з набору об’єктів, кожен з яких характеризується станом пам’яті та поведінкою. Об’єкти взаємодіють між собою, використовуючи для цього повідомлення.

Будується ієрархія об'єктів: програма загалом – об'єкт; для виконання своїх функцій вона звертається до інших об'єктів, що містяться у ньому, які водночас виконують запити шляхом звернення до інших об'єктів програми.

Звісно, щоб уникнути безкінечної рекурсії у зверненнях, на якомусь етапі об'єкт трансформує запит у повідомлення до стандартних системних об'єктів, які даються мовою та середовищем програмування.

Стійкість та керованість системи забезпечуються через чіткий розподіл відповідальності об'єктів (за кожну дію відповідає певний об'єкт), однозначного визначення інтерфейсів міжоб'єктної взаємодії та повної ізольованості внутрішньої структури об'єкта від зовнішнього середовища (інкапсуляції).

Оскільки саме принципи ООП були засадничими під час розроблення мови C++, тому важливо чітко визначити, чим вони є. Отож ООП об'єднало кращі ідеї структурного програмування з рядом потужних концепцій, які сприяють ефективній організації програм.

Об'єктно-орієнтований підхід до програмування дає змогу розкласти задачу на складові так, що кожна частина стане самостійним об'єктом, який містить власні дані та методи для роботи над ними. За такого підходу істотно знижується загальний рівень складності написання кодів програм, що дає змогу програмісту справлятися зі значно складнішими програмами.

Всі мови ООП характеризуються трьома основними ознаками: інкапсуляцією, поліморфізмом і успадкуванням. Розглянемо стисло кожну з них.

2.5.2. Поняття про механізм реалізування програм методом інкапсуляції

Як відомо, усі програми здебільшого складаються з двох основних елементів: методів (кодів програм) і даних. Код – частина програми, яка реалізовує дії, а дані є інформацією, до якої застосо-

вуються ці дії. Інкапсуляція – такий механізм програмування, який пов'язує в одне ціле програмні коди і дані, які вони обробляють, щоб забезпечити їх як від зовнішнього втручання, так і від неправильного використання.

У об'єктно-орієнтованій мові програмування коди і дані можуть пов'язуватися між собою так, що створюється чорний ящик. У цьому "ящику" містяться всі необхідні (для забезпечення самостійності) дані та коди. При такому взаємозв'язку кодів програми і даних створюється об'єкт, тобто об'єкт – конструкція, яка підтримує інкапсуляцію.

Усередині об'єкта програмні коди, дані або обидві ці складові можуть бути не тільки відкритими (*public*) або закритими (*private*), але й захищеним (*protected*). Закритий програмний код (або дані) відомий і доступний тільки іншим частинам того ж об'єкта, тобто до нього не може отримати доступ та частина програми, яка існує поза цим об'єктом. Відкритий програмний код (або дані) доступний довільним іншим частинам програми, навіть якщо вони визначені в інших об'єктах. Захищений програмний код (або дані) буде доступним для інших елементів програми, які не є членами цього об'єкта, за винятком того, що доступ до захищеного члена є ідентичним доступу до закритого члена, тобто до нього можуть звертатися тільки інші члени того ж об'єкта. Зазвичай відкриті частини об'єкта використовують для надання керованого інтерфейсу із закритими елементами об'єкта.

2.5.3. Поняття про властивість поліморфізму

Поліморфізм (від грецького слова *polymorphism*, що означає "багато форм") – властивість, яка дає змогу використовувати один інтерфейс для цілого класу дій. Конкретна дія визначається характерними ознаками ситуації.

Як простий приклад поліморфізму можна навести кермо автомобіля. Для керма (тобто інтерфейсу) байдуже, який тип

рульового механізму використовується в автомобілі. Тобто, кермо працює однаково, незалежно від того, чи оснащений автомобіль рульовим керуванням прямої дії (без підсилювача), рульовим керуванням з підсилювачем або механізмом рейкової передачі. Якщо Ви знаєте, як поводитися з кермом, тоді Ви зможете кермувати автомобілем довільного типу.

Цей же принцип можна застосувати до програмування. Розглянемо, наприклад, стек, або список, додавання елементів до якого або видалення з нього здійснюється за принципом "останній прибув – перший обслужений". У Вас може бути програма, у якій використовуються три різних типи стека. Один стек призначений для цілочисельних значень, другий – для значень з плаваючою крапкою і третій – для символів.

Алгоритм реалізуванні всіх стеків – один і той самий, хоча у них зберігаються дані різних типів. У не об’єктно-орієнтованій мові програмісту довелося б створити три різні набори підпрограм обслуговування елементів стека, причому підпрограми мали б мати різні імена, а кожен набір – власний інтерфейс. Але завдяки поліморфізму у мові С++ можна створити один загальний набір підпрограм, який підходить для всіх трьох конкретних ситуацій. Отож, знаючи, як використовувати один стек, програміст може використовувати й усі інші.

У більш загальному вигляді концепція поліморфізму відтворюється фразою "один інтерфейс – багато методів". Це означає, що для групи взаємопов’язаних дій можна використовувати один узагальнений інтерфейс. Поліморфізм дає змогу знизити рівень складності через можливість застосування одного і того ж інтерфейсу для виконання цілого класу дій. Вибір же конкретної дії (тобто функції) стосовно тієї або іншої ситуації лягає "на плечі" компілятора. Вам, як програмісту, не потрібно робити цей вибір вручну. Ваше завдання – використовувати загальний інтерфейс.

Перші мови ООП були реалізовані у вигляді інтерпретаторів, тому поліморфізм підтримувався у процесі виконання програм. Проте мова програмування С++ – трансльована мова. Отже, у мові С++ поліморфізм підтримується на рівні і компілювання програми, і її виконання.

2.5.4. Поняття про використання процесу успадкування

Успадкування – це процес, завдяки якому один об'єкт може набувати властивості іншого. Завдяки успадкуванню підтримується концепція ієрархічного класифікування. У вигляді керованого ієрархічного (низхідної) класифікування організовуються більшість областей знань. Наприклад, яблука Джонатан (йони) є частиною класифікування яблук, яка сама є частиною класу фрукти, а той – частиною ще більшого класу їжа.

Отже, клас "їжа" володіє певними якостями (їстівність, поживність тощо), які застосовуються і до підкласу "фрукти". Окрім цих якостей, клас "фрукти" має специфічні характеристики (соковитість, солодкість), які відрізняють їх від інших харчових продуктів. У класі "яблука" визначаються якості, специфічні для них (ростуть на деревах, не тропічні). Клас "Джонатан" успадковує якості всіх попередніх класів і водночас визначає якості, які є унікальними для цього сорту яблук.

Якщо не використовувати ієрархічного подання ознак, для кожного об'єкта довелося б у явній формі визначити всі притаманні йому характеристики. Але завдяки процесу успадкування об'єкту потрібно додатково визначити тільки ті якості, які роблять його унікальним усередині його класу, оскільки він (об'єкт) успадковує загальні атрибути свого батька. Отже, власне механізм успадкування дає змогу одному об'єкту представляти конкретний примірник більш загального класу.

2.6. Програмні середовища

Від складання програмістом до виконання комп'ютером програма проходить доволі тривалий процес оброблення спеціальними службовими програмами, які становлять систему автоматизації програмування. З часом слово "автоматизація" випало із поданого словосполучення, внаслідок чого воно перетворилося на "систему програмування".

Така система складається з кількох компонент, а саме препроцесора (*preprocessor*), компілятора (*compiler*) та компонувальника (*linker*), а також засобів підтримання етапу виконання, зокрема налагоджувача (*debugger*), об'єднаних спільним інтерфейсом у так зване універсальне середовище розроблення програм.

Прикладами таких середовищ можуть бути система програмування Visual C++ та діалогове середовище розроблення програм Developer Studio. Головна особливість підготування програми до виконання полягає в тому, що програму збирають із багатьох, іноді різнорідних, частин, об'єднаних у програмний проєкт *(software project)*. Кожну програмну розробку оформлюють як окремий проєкт, який зазвичай складається з багатьох різних файлів.

Файли першого типу називають вхідними (*source file*, початковий файл), вони містять тексти, написані певною мовою програмування. Робота над проєктом може бути достатньо тривалою, в її процесі виникатимуть нові вхідні файли, які додаватимуться до наявних. Тому в кожен момент частина вхідних файлів може виявитися вже готовою до використання, тобто попередньо відкомпільованою.

Цю частину використовують у вигляді готових машинних кодів, що зберігаються в проєкті як особливі об'єктні файли (*object file*), котрі складають другий тип файлів програмного проєкту. Система програмування автоматично відстежує необхідність повторного компілювання кожного вхідного файлу. Компілюванню підлягає кожен новий файл, долучений до проєкту, а також довільний інший файл після внесення до нього змін.

Крім спеціально розроблених власних кодів, проєкти можуть використовувати стандартне програмне забезпечення, яке зберігається в системних бібліотеках, передбачена також можливість створення і подальшого використання власних бібліотек. C++ послідовно продовжує закладену ще авторами мови C тенденцію широкого використання програмних бібліотек. Зокрема, мовою не визначено ніяких способів зв'язку програми з операційною системою – ці функції повністю перекладено на системні бібліотеки.

Вже упродовж не одного десятиріччя під час створення великих за розмірами програм використовують принцип структурної

декомпозиції. Логічним складовим програми (у процедурному програмуванні це процедури і функції, у більш розвинених парадигмах – модулі або класи) відповідають фізичні складові – файли, які містять ці окремі завершені логічні частини, одну або декілька разом.

Одночасне використання в одному програмному проєкті багатьох файлів з кодами різних частин програми одержало назву роздільного компілювання (*separate compilation*), яке, без сумніву, стало одним із найбільших досягнень систем програмування.

Програмний код тепер поділено на частини, які називаються одиницями транслювання (*translation unit*). З використанням роздільного компілювання стало можливим колективне розроблення великих програм: окремі розробники залучені кожен до своїх файлів.

Програмування – це діяльність, яка потребує великої організованості. Прийнято не тільки розділяти текст програми на структурні частини, але й розрізняти описові та виконавчі частини кодів.

Тому, під час створення програмних кодів дотримуються певних правил доброго тону, одне з яких полягає в необхідності розподілення визначень і обчислень між вхідними файлами двох типів: файлами заголовків *(header)* і файлами реалізування *(implementation)*.

Особливий компонент системи програмування – препроцесор – відповідно до директив, розміщених здебільшого, на початку вхідного файлу, приєднує до нього відповідні файли заголовків.

Вони містять інформацію про те, які функції, об’єкти або класи, визначені поза цим вхідним файлом, можуть бути використані в ньому. Один файл заголовків можна приєднувати до багатьох вхідних файлів, які використовуватимуть оголошені в ньому конструкції. Приєднанням потрібних заголовних файлів препроцесор готує вхідні файли до подальшого оброблення компілятором. Тому вміст вхідного файлу для компілятора, загалом кажучи, відрізняється від вмісту вхідного файлу, створеного розробником, оскільки оброблення препроцесором може значно його змінити. Далі розглянемо, в яких межах варто зосередити це оброблення.

Набір пов’язаних один з одним заголовних і вхідних файлів, з яких складається проєкт, сукупно утворює вхідну програму, яку система програмування перетворюватиме у машинний код. Машинний код генерується у два етапи. На першому компілятор

генерує так звані об’єктні коди (*object code*), по одному для кожного файлу. Коди називають об’єктними (від слова *"objective"*, що англійською мовою означає "мета"), оскільки їхнє створення є метою роботи компілятора, який становить центральну частину системи програмування.

Інша її частина, названа компонувальником (*linker*), збирає об’єктні файли в одну виконувану програму (*executable program*, *exe-file*), приєднуючи до неї також об’єктні коди зі стандартних або власних бібліотек проєкту.

Ми розглядатимемо систему програмування такою мірою, наскільки вона впливає на створення та організацію кодів вхідних програм.

За детальним ознайомленням із засобами і можливостями систем програмування відсилаємо до системної документації та спеціальної літератури.

Ось типовий сценарій підготовки програми. Система програмування працює з програмними проєктами, де розміщують файли вхідної програми. Розглянемо, наприклад, проєкт `sqrt`. До файлу реалізуванні `root.cpp` запишемо програмний код функції `root` для обчислення квадратного кореня, а програмний код для її виклику помістимо у файл реалізування `main.cpp`.

Пам’ятаймо, що компілювання цих двох файлів виконуватиметься окремо. Виникає запитання: звідки під час компілювання файлу `main.cpp` компілятор візьме відомості про ідентифікатор `root`? Загалом можливі два розв’язання. Перше – це перемістити код функції з файлу `root.cpp` до файлу `main.cpp`, де функцію використано. Проте це було б не дуже вдалим виходом, оскільки незрозуміло, як одну й ту ж функцію розмістити у різних вхідних файлах, де її також можна було б викликати.

Друге розв’язання, а саме воно прийняте в системах програмування, полягає у створенні допоміжного заголовного файлу `root.h`, який міститиме не всю інформацію про функцію `root`, а лише ту, яка необхідна для оброблення її виклику компілятором, – так звану сигнатуру, прототип функції:

```
// root.h
double root(double x, double eps);
```

Проєкт sqrt, наповнений вхідними файлами, подано на рисунку 2.3. Заголовний файл приєднують до вхідного за допомогою команд препроцесора. Їх записують за допомогою спеціальних директив, які починаються символом #.

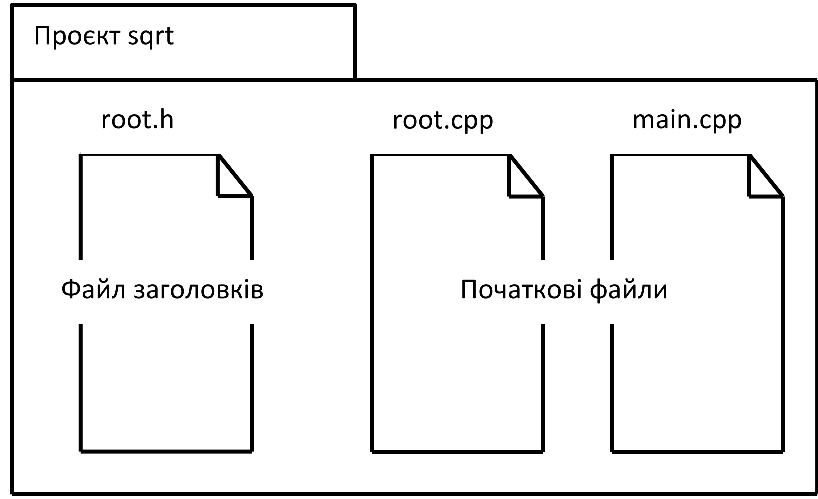


Рис. 2.3. Склад вхідних файлів проєкту sqrt

Зокрема, файл `main.cpp` міститиме директиву `#include "root.h"`, виконання якої препроцесором полягає у включенні тексту з файлу заголовку `root.h` до вхідного файлу `main.cpp`. Подамо код, доповнений коментарями, призначеними зробити його зрозумілим навіть для читача, який ще зовсім не знає C++.

```
// main.cpp
// Спочатку системні заголовки
// Під'єднння бібліотеки введення та виведення
#include <iostream>
// Під'єднння бібліотеки математичних функцій
#include <cmath>
// Посилання на стандартні позначення
using namespace std;
```

```
// Тепер власні заголовки
#include "root.h"
// Початок власне програми, вона називається
// main
int main() {
// cout<<someting; вивести something
// cout<<someting<<endl; вивести те ж саме і
// перейти на новий рядок
cout<<"The square root of 2 calculated with"<<
endl<<" a standart fuction is"<<sqrt(2.0)
<<endl;
cout<<" your function is "<<root
(2.0,0.000001)<<endl;
return 0;
}
```

Як бачимо, крім заголовного файлу `root.h`, вхідний файл використовує ще кілька інших, службових файлів, приєднаних препроцесором (їх не показано на рисунку 2.3).

Файли `iostream` і `cmath` описують стандартні бібліотеки: введення/виведення `iostream` (вона потрібна для обслуговування потоку виведення `cout`) та математичних функцій `cmath`, звідки буде взято, наприклад, функцію обчислення квадратного кореня `sqrt()`.

По суті функція `sqrt()` не є частиною мови програмування C++, але її "знає" кожен C++-компілятор. Крім бібліотеки функцій, кожен C++-компілятор також містить бібліотеку класів, яка є об’єктно-орієнтованою бібліотекою. Нарешті, у мові програмування C++ визначено стандартну бібліотеку шаблонів (*Standard Template Library – бібліотека STL*). Вона надає процедури "багаторазового використання", які можна налаштовувати відповідно до конкретних вимог. Але, перш ніж застосовувати бібліотеку класів або бібліотеку STL, нам необхідно познайомитися з класами, об’єктами і зрозуміти, у чому полягає суть шаблона.

Директива `using namespace std` є вказівкою компілятору вживати стандартні імена для системних об’єктів: наприклад, через `cout` позначено стандартний потік виведення.

Відповідного доповнення потребуватиме також файл `root.cpp`, у якому використано стандартну функцію `fabs()` обчислення абсолютної величини дійсного числа:

```
// root.cpp
#include <cmath>
double root (double x, double eps)
{
double s=0.5*x;
double t;
do {
t=s; s=(s+x/s)*0.5;
}
while ((fabs(s-t)/s)>eps);
return s;
}
```

Тепер коротко про призначення інших компонентів системи програмування. Після того, як буде виконане інтепретування команди препроцесора, файли повних текстів надійдуть на вхід компілятора, який створить з них об’єктні коди (рисунок 2.4).

Об’єктний код ще не призначено для безпосереднього виконання. У проєкті таких кодів багато, загалом, по одному на кожен вхідний файл, і вони містять взаємні посилання один на одного, а також на бібліотеки. Об’єктні коди надходять на вхід компонувальника, завдання якого – зібрати їх разом, доповнити необхідними компонентами бібліотек та перетворити на виконувану програму.

На вимогу розробника компілятор може доповнити об’єктні коди спеціальними командами спостереження за ходом виконання програми. Тоді, властиво, самим виконанням програми займеться налагоджувач (*debugger*), який дасть змогу простежити за перебігом цього процесу, призупинити його, проконтролювати значення змінних тощо.

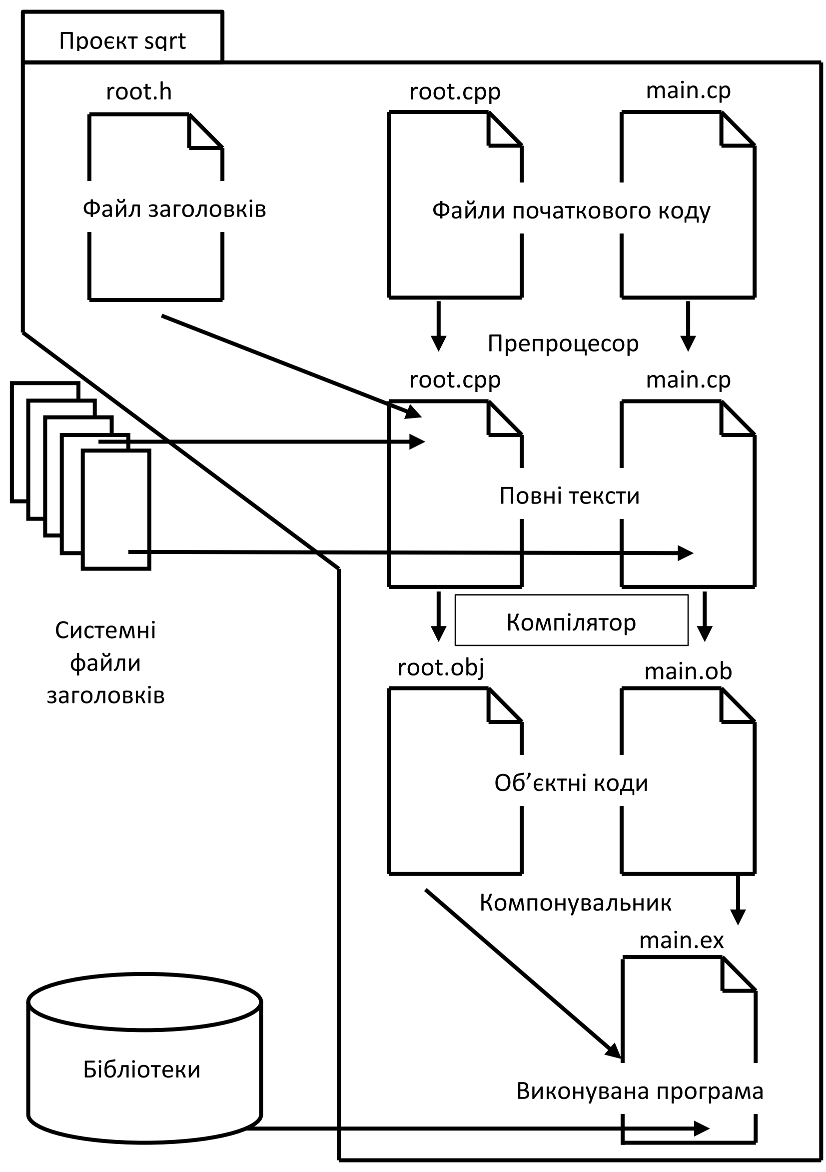


Рис. 2.4. Повний склад файлів проєкту `sqrt`

У читачів цього посібника, які щойно розпочали вивчення програмування, може виникнути сумнів у потрібності матеріалу, викладеного у межах цього розділу. Проте з подальшим опануванням мови C++ більшість повернетяся до повторення понять, які тут викладені.

Розділ
3

БАЗОВІ ЕЛЕМЕНТИ C++

3.1. Склад C++

У звичайних мовах спілкування поміж людьми виокремлюють чотири основні елементи: символ, слово, словосполучення та речення. Подібні елементи існують і в алгоритмічній мові, однак слова називають лексемами (елементарні конструкції), словосполучення – виразами, а речення – настановами. Лексеми створюються із символів, вирази – із лексем та символів, настанови – з символів, виразів і лексем:

– алфавіт мови, або її допустимі для використання символи – це основні неподільні знаки, з використанням яких створюють програмні коди мовою програмування;

– лексема, або елементарна конструкція – це елементарна одиниця мови програмування, яка має самостійний зміст;

– вираз визначає правило обчислення деякого значення;

– настанова визначає скінченний опис деякої дії.

Для опису мови програмування у документації часто використовують деяку формальну метамову, наприклад, формули Бекуса-Наура або синтаксичні діаграми. Ми застосовуємо розповсюджений неформальний спосіб опису, який передбачає необов'язкові частини синтаксичних конструкцій записувати у квадратних дужках, текст, який необхідно замінити конкретним значенням, записуватимемо українською мовою, а вибір одного з кількох елементів позначається вертикальною рискою, наприклад, запис

```
[ void | int ] ім'я()
```

означає, що замість конструкції `ім'я` необхідно вказати конкретне ім'я відповідно до правил алгоритмічної мови, а перед ним може

знаходитися або `void`, або `int`, або взагалі нічого. Фігурні дужки використовуються для групування елементів, з яких необхідно обрати тільки один. Якщо квадратні дужки будуть використовуватися як елемент синтаксису, про це буде сказано окремо.

Алфавіт мови С++ охоплює:

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

Інші символи, а також символи кирилиці не використовуються для побудови базових елементів мови або для їх розділення, але вони можуть застосовуватись у символьних константах та коментарях.

З символів алфавіту формуються **лексеми** мови:

– ідентифікатори;
– ключові (зарезервовані) слова;
– оператори (знаки операцій);
– константи;
– розділювачі (дужки, крапка, кома, символ пробілу).

Межі лексем визначаються іншими лексемами-розділювачами або операторами.

3.2. Літерали та їхні типи

Літерал (від лат. *literal* – буквальний) – це найпростіший об’єкт, який позначає сам себе. Літерал не потрібно попередньо оголошувати або визначати. Тому, коли він трапляється у програмному коді, компілятор має самостійно розпізнати його тип і визначити для його зберігання пам’ять відповідного розміру.

3.2.1. Числові літерали

Потреба розпізнавати тип за записом літералу призводить до певних умовностей: наприклад, у мові C літерали 10 і 010 різняться своїми значеннями, тому що вони позначають різні числа – 10 і 8. Водночас літерал 08 компілятор сприйме як помилку. Річ у тім, що за традицією, успадкованою від мови C, числові константи різняться за такими системами числення:

– десятковою;
– вісімковою;
– шістнадцятковою (хоча це стосується лише цілих констант).

Двійковий вміст машинного слова простіше відтворити за шістнадцятковим записом числа: один байт відповідає двом шістнадцятковим цифрам, півслово – чотирьом, слово – восьми. Використання вісімкового коду зумовлене ще давнішою традицією: раніше було прийнято називати байтом 9 бітів, а це рівно три вісімкові цифри. Зазначимо, що знову йдеться про програмування на низькому рівні.

Цілі числа у вісімковому записі починаються нулем (математична традиція нехтування передніми нулями у програмуванні мовою C++ не діє). Це, мабуть, пояснюється тим, що нуль нагадує літеру "O" (від *octal* – вісімковий, скорочено *oct*).

Шістнадцятковий запис цілого числа починається символами 0x (цифрою "0" і літерою "x", яка має нагадувати слово *hexadecimal* – шістнадцятковий, скорочено *hex*). Нижче подано три різні записи числа 100:

```
100  // десятковий запис
0144 // вісімковий
0x64 // шістнадцятковий
```

Вісімковими та шістнадцятковими константами потрібно користуватися лише для типів без знаку, щоб на комп’ютерах з доповнювальним кодом (*complement-on-two*), який зазвичай використовують для подання від’ємних чисел, не стикатися з несподіванками

на кшталт значення –1 константи 0xffff (для 16-бітового слова) або 0xffffffff (для 32-бітового).

Можна уточнити тип цілого літералу, додавши до його запису U – для беззнакового, L – для довгого, UL – для беззнакового довгого типу, наприклад, записи 2147483648U та 2147483648UL позначають число 231.

Спроба ж розглянути 2147483648 як число зі знаком, скажімо *signed long*

```
int surprise=2147483648UL;
cout<<surprise<<endl;
```

може мати доволі несподівані наслідки.

Дійсні літерали вважають сталими з подвійною точністю. На звичайну точність вказує F, а на подвійну – L, наприклад 1.0F або 3.1415926535897932L.

3.2.2. Символьні літерали

У мові C++ розрізняють одно- та багатосимвольні літерали, і вони мають різне машинне кодування. Односимвольний літерал (надалі називатимемо його просто символьним) завжди одержує тип `char`. Зазвичай (але не обов’язково) це один байт. Область пам’яті, зайняту символом, вважають одиницею пам’яті.

Для позначення символьної константи її записують в одинарних лапках. Це може бути символ або його код. Ось приклади явно заданих символів.

```
// Звичайні символи
'?', 'f', 'ї'
```

Якщо символ задають його кодом, зокрема це стосується символів керування, тоді коду символу передує знак зворотної скісної риски (*backslash*) ‘\’. Символ може бути закодовано вісімковим або шістнадцятковим кодом.

Подамо приклади вісімкових кодів:

```
// Вісімкові коди
'\77'  // символ '?'
'\144' // символ 'f'
'\365' // символ 'ї'(з точністю до кодової
       // таблиці)
'\11'  // символ табуляції
'\12'  // символ нового рядка
'\14'  // символ нової сторінки
```

Літера `'x'` після зворотної скісної риски свідчить про застосування шістнадцяткового коду:

```
// Шістнадцяткові коди
'\x3f' // символ '?'
'\x66' // символ 'f'
'\x10' // символ нового рядка
```

Крім того, існують особливі позначення для деяких службових символів, наприклад,

```
// Службові символи
'\t' // символ табуляції
'\n' // символ нового рядка
'\f' // символ нової сторінки
```

Як бачимо, деякі символи можна записати еквівалентними способами:

```
'\x10'; '\12'; '\n'; // символ нового рядка
'f'; '\144'; '\x66'  // символ нової сторінки
```

Багатосимвольні літерали – рядки символів – записують у подвійних лапках. Вони зберігаються в пам’яті як послідовності символів, що закінчуються нульовим кодом `'\0'`. За такого способу кодування не потрібно пам’ятати довжину рядків: діставшись

першого від початку нульового коду, потрапляємо в кінець рядка. Тому довжина кожного рядка на одиницю більша за кількість його символів. Отже, порожній рядок містить один символ – нульовий код.

3.3. Константи та змінні

Як і в інших мовах програмування, у С++ найчастіше використовують іменовані об'єкти – константи (сталі) та змінні. На відміну від літералів, іменовані об'єкти потрібно попередньо визначати, задаючи тип об'єкта та у більшості випадків (а для сталих величин – завжди) його значення. Значення змінних можуть змінюватись у процесі виконання програми, а значення констант залишаються незмінними і їх незмінність контролює компілятор.

3.3.1. Константи

Константами (*constant*) називають незмінні значення. Розрізняють цілі, дійсні, символьні та рядкові константи. Компілятор, зазначивши константу як лексему, відносить її до одного з типів за зовнішнім поданням. Константи також називають літералами *(literals)*.

Характерна особливість їх визначень порівняно з визначеннями змінних – наявність специфікатора (*qualifier*) константи `const`, наприклад:

```
const double pi=3.14159F;
```

Спосіб визначення кожної константи залежить від її типу. Константи мови С++ потрібно поділяти на *літеральні* та *типізовані.*

Викладаючи цей матеріал, автори дещо повторюватимуться для кращого розуміння, особливо початківцями.

Літеральна константа – це лексема, яка є зображенням фіксованого числового, рядкового або символьного значення. Такі константи бувають цілі, дійсні, символьні та рядкові (див. табл. 3.1).

Таблиця 3.1

Літеральні константи мови С++

Константа	Формат	Приклади
Ціла	Десятковий: послідовність десяткових цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), що починається не з нуля, якщо це не число нуль	9, 0, 217925
	Вісімковий: нуль, за яким розташовані вісімкові цифри (0, 1, 2, 3, 4, 5, 6, 7)	02, 050, 07245
	Шістнадцятковий: 0x або 0X, за яким розташовані шістнадцяткові цифри (0, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)	0x1B9, 0X00FF
Дійсна	Десятковий:[цифри].[цифри] Експоненційний: [цифри][.][цифри]{E|e}[+|-][цифри]	9.7, .001, 87. 0.7E6, .15e-3 9.2, 920 e-2, 92.E-1, .92E1
Символьна	Один або два символи, що подаються в апострофах	'A', 'ю', '*', '\0', '\n', '\012', '\x07\x07'
Рядкова	Послідовність символів, що подаються в лапках	"RESULT", "\t sum__s=\0x5\n"

Цілі константи можуть бути десятковими, вісімковими та шістнадцятковими.

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

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

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

Експоненційна форма дійсного числа використовується для запису дуже великих або дуже малих чисел, для яких задавати зайві нулі не зовсім зручно, наприклад, 1 .0123*10^{20},1.0123*10^{-10}. У цій формі запису числа можна виокремити такі основні характеристики: знак числа, мантису числа, знак порядку та порядок числа. Зазначені характеристики дійсного числа зберігаються у пам'яті комп'ютера. Число у показниковій формі може бути подано, наприклад, так: 1.0123E-10. Мантиса записується ліворуч від знака експоненти (E або e), порядок – праворуч. Символ E (e) означає основу степеня 10, і компілятор розпізнає цей запис як форму подання дійсного числа. Символ пропуску всередині числа не допускається, а для відділення цілої частини від дробової використовується не кома, а крапка. При додатних значеннях числа і мантиси знак "+ " можна не вказувати.

І десяткова, і експоненційна форми запису допускають відсутність або цілої частини, або дробової, але не двох одразу.

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

Зображення від'ємної цілої або дійсної константи вважається константним виразом, що складається зі знака унарної операції зміни знака (-) та константи, наприклад: -273, -2730.e-1, -273L.

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

Заслуговують уваги послідовності, що починаються зі знака "\", їх називають *керуючими* або *escape*-послідовностями. Символ зворотної скісної риски "\" (*backslash*) використовується

для запису кодів, що не мають графічного зображення, для запису символів, а також для виведення символьних констант, якщо їх коди задані у вісімковому та шістнадцятковому вигляді (див. табл. 3.2).

Таблиця 3.2

Керуючі послідовності мови С++

\a	звуковий сигнал
\b	повернення на крок
\f	переведення сторінки (формату)
\n	новий рядок
\r	повернення каретки
\t	горизонтальна табуляція
\v	вертикальна табуляція
\\	символ \ – зворотна коса риска
\′	символ ' – апостроф
\″	символ " – лапки
\0	нуль-символ
\?	знак запитання
\0ddd	вісімковий код символу
\0xddd	шістнадцятковий код символу

Символьні константи зображують значення типу `char`, які займають один байт. Цей байт як зображення числа без знака дає код символу (число від 0 до 255), а зі знаком – від –128 до 127.

Відповідність між числами від 0 до 127 і символами зафіксовано в *Американському стандартному коді для обміну інформацією* (так звана *таблиця ASCII*), решті кодів можуть відповідати різні набори символів. Наприклад, константа `' '` (пробіл) має код 32, а `'\0'` – код 0. Окремими властивостями таблиці ASCII є такі:

– символам '0', '1', …, '9' відповідають послідовні коди від 48 до 57;

– символам 'A', 'B', …, 'Z' – від 65 до 90;

– символам 'a', 'b', …, 'z' – від 97 до 122.

Настанова cout<<'Z'; виводить значення константи 'Z' – на екрані з'являється Z. Аналогічно можна вивести інші символьні константи. Для того, щоб наступне повідомлення виводилося з нового рядка, можна скористатися настановами cout<<endl; або cout<<'\n';.

Настановою cout<<*константа*; обробляється не *константа*, а зображене нею значення. Послідовність символів, утворена за значенням, може відрізнятися від константи.

Приклад 3.1:

```
#include <iostream>
using namespace std;
int main() {
   cout<<'1'; cout<<'\n';
   cout<<'Y';
   cout<<'e';
   cout<<'s'; cout<<endl;
   system("pause");
return 0;
}
```

Рядкова константа (рядковий літерал) – це послідовність символів, яка подається в лапках (тобто в символах ") і зберігається у неперервній ділянці пам'яті, наприклад: "Це рядковий літерал".

Керуючі послідовності можуть також застосовуватись у рядкових константах. Так, якщо всередині рядка потрібно записати лапки, тоді перед ними слід розташувати зворотну скісну риску ("\"), за якою компілятор відрізняє їх від лапок, що обмежують рядок:

```
"Курс лекцій \"Алгоритмізація та програмування (C++)\" "
```

Рядки, які записані у програмі поспіль або через символи пропуску, при компілюванні конкатенуються (склеюються). Тобто послідовність двох рядків

```
"Світячи іншим, згораєш сам."
"Успіх – це встигнути."
```

цілком еквівалентна рядку:

```
"Світячи іншим, згораєш сам.
Успіх – це встигнути."
```

Довгу рядкову константу можна розмістити також у декількох рядках. У цьому разі ставиться зворотна скісна риска і натискається клавіша *Enter*. Наприклад:

```
"Програма виконує те, \
Що Ви їй вказали виконувати, але не те, \
що Ви хотіли, щоб вона виконувала. "
```

Використовуючи рядкові константи, можна записати так:

```
#include <iostream>
using namespace std;
int main() {
cout<<"Програма виконує те,\n що Ви їй вказали виконувати, але не те,\n що Ви хотіли, щоб вона виконувала.";
system("pause");
return 0;
}
```

3.3.2. Змінні

Кожна програма потребує виконання різноманітних обчислень, для здійснення яких використовуються вирази, які складаються з операндів, операторів (символів операцій) і дужок.

Операнди визнають дані для обчислень, а оператори визнають дії, які необхідно виконати над цими даними. Операнд є, своєю чергою, виразом, що в окремому випадку може бути константою або змінною.

Поняття змінної числової величини увів у своїх працях Декарт ще у XVII ст. Пізніше з'ясувалося, що змінна величина може бути не лише числовою. У найширшому розумінні змінна величина – це загальне поняття реального або уявного об'єкта (або його окремої характеристики), який може перебувати в різних станах.

Змінні величини позначають символічними іменами або просто іменами. У програмуванні змінна є моделлю (зображенням) об'єкта в пам'яті комп'ютера.

Змінна в машинній програмі – це ділянка пам'яті, яка може мати різні стани й зображувати ними різні стани об'єкта.

Змінну у програмному коді мовою високого рівня позначає ім'я. Кожна ділянка пам'яті комп'ютера має адресу (номер першого байта ділянки), і цією адресою позначається у машинній програмі. Кажуть, що адреса вказує, або посилається на ділянку. Під час транслювання програмного коду ім'я змінної перетворюється на адресу деякої ділянки пам'яті. Отже, вважатимемо, що під час виконання програми ім'я змінної вказує на ділянку пам'яті.

Станам об'єкта, зображуваного змінною, відповідають її значення (числові та інші), які належать до певного типу. Тип змінної та її ім'я задаються настановою оголошення імені змінної, що має вигляд:

```
<ім'я_типу> <ім'я_змінної>;
```

Після імені типу можна записати кілька імен змінних, відділяючи їх комами. Настанова *оголошення* імені змінної одночасно є *означенням* змінної, тобто задає ділянку пам'яті, на яку вказує ім'я змінної.

Отже, підсумовуючи вказане, будемо вважати, що *змінна* – це іменована область пам'яті, у якій зберігаються дані визначеного типу. Змінна має ім'я, розмір та інші атрибути, такі як область видимості, час існування тощо. Ім'я змінної слугує для звертання до області пам'яті, у якій зберігається її значення. Перед

використанням довільна змінна повинна бути оголошена, водночас для неї резервується деяка область пам'яті, розмір якої залежить від конкретного типу змінної. Під час виконання програми змінна може приймати різні значення.

У програмному коді змінна позначається ідентифікатором (символічним іменем). За ідентифікатором здійснюється звернення до значення змінної.

Ідентифікатором, тобто ім'ям програмного об'єкта, називається довільна послідовність літер латинського алфавіту, цифр і символу підкреслення за умови, що першою стоїть літера або символ підкреслення, а не цифра. Великі та малі літери розрізняються.

Існує два різновиди ідентифікаторів:

– *стандартні*, наприклад, імена всіх вмонтованих у мову C++ функцій;

– *користувача*.

Характерно, що мова C++ чутлива до регістру літер, тому компілятор розпізнає великі та малі літери латинського алфавіту як різні символи. Це дає можливість створювати ідентифікатори, які однаково читаються, але відрізняються написанням одного або декількох символів. Наприклад, ідентифікатори `ALFA` і `Alfa`, `sigma` і `sigMa` вважаються різними.

Стандарт не обмежує довжини ідентифікатора і він може мати довільну довжину, але значимими є не більше 31 символів від початку ідентифікатора, а у деяких компіляторах це обмеження ще більш суворе (не більше, як 8 символів). Ідентифікатор створюється на етапі оголошення змінної, функції, після чого його можна використовувати у настановах програмного коду.

Надамо декілька порад щодо вибору ідентифікатора:

– рекомендується не жалкувати часу на створення ідентифікаторів здебільшого за їх змістовим призначенням;

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

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

– не слід починати ідентифікатори із символу підкреслення, бо вони можуть збігатися з іменами системних функцій або змінних;

– для розділення частин імені можна використовувати символ підкреслення;

– всередині ідентифікатора не можна розміщати символи пропуску.

Приклад опису цілої змінної з іменем `a` і дійсної змінної `x`:

```
int a; float x;
```

Тепер подамо загальний формат опису змінних:

<тип_змінної> <ім'я_змінної> [=ініціалізатор];

Також у іншій формі запису:

[клас пам'яті] [const] <тип> <ім'я> [=ініціалізатор];

де:

– необов'язковий `клас пам'яті` може приймати одне зі значень – `auto, extern, static` або `register` (у посібнику при описанні синтаксису об'єктів програмування модифікатор `const` вказує, що змінна не може змінювати своє значення, у цьому разі її називають типізованою (іменованою) константою або просто константою);

– `ініціалізатор` – це присвоювання змінній при оголошенні початкового значення, яке записується з символом дорівнює – `= значення` або в круглих дужках – `(значення)`.

Зазначимо, що константа повинна бути ініціалізована при оголошенні. Одна настанова може містити оголошення декількох змінних одного типу, відділяючи їх комами, наприклад:

```
short int a = 1; // ціла змінна a
const char C = 'C';// символьна константа C
char s,sf = 'f'; // ініціалізування відноситься
                 // тільки до sf
char t(54);
float c = 0.22, x(3), sum;
extern int x;
```

Опис змінної може виконуватися у формі оголошення або визначення (ініціалізування). Оголошення інформує компілятор про тип змінної та класи пам’яті, а визначення містить, крім того, вказівку компілятору про виділення пам’яті відповідно до типу змінної.

У С++ більшість оголошень є одночасно і визначеннями (у поданому програмному фрагменті тільки опис

```
extern int x;
```

є оголошенням, але не визначенням).

Змінна може бути оголошена багаторазово, а визначена тільки в одному місці програми, оскільки оголошення тільки описує властивості змінної, а визначення зв’язує її з конкретною областю пам’яті.

Якщо тип значення, яке ініціалізується, не збігається з типом змінної, тоді виконуються перетворення типу. Кожна змінна повинна мати своє ім’я, причому в одному блоці не може бути двох змінних з однаковим ім’ям.

Опис змінної, окрім типу і класу пам’яті, явно або за замовчуванням задає її область дії. Клас пам’яті й область дії залежать не тільки, властиво, від опису, але і від місця розташування у тексті програми.

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

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

Клас пам’яті визначає час існування та область видимості програмного об’єкта, тобто змінної. Якщо клас пам’яті не визначений явно, тоді він визначається компілятором, виходячи з контексту оголошення.

Час існування (часом у літературі – життя) змінної може бути постійним (протягом виконання програмного коду) і тимчасовим (протягом виконання блока).

Областю видимості ідентифікатора називається частина програмного коду, з якого можна здійснити звичайний доступ до зв'язаної з ідентифікатором області пам'яті. Найчастіше область видимості збігається з областю дії. Винятком є ситуація, коли у вкладеному блоці описана змінна з таким же ім'ям. У цьому разі зовнішня змінна у вкладеному блоці невидима, хоча він і входить до її області дії. Проте до цієї змінної, якщо вона глобальна, можна звернутися, застосовуючи оператор доступу до області видимості – `"::"`.

Клас пам'яті задають такі специфікатори:

– `auto` – автоматична змінна, для якої пам'ять виділяється у стеку і за необхідності ініціюється кожного разу при виконанні настанови, яка містить її визначення. Звільнення пам'яті відбувається при виході з блока, де описана змінна. Час її існування – з моменту опису до кінця виконання блока. Для глобальних змінних цей специфікатор не використовується, а для локальних він приймається за замовчуванням, тому задавати його явно великого сенсу немає;

– `extern` означає, що змінна визначена в іншому місці програми (в іншому файлі або далі за текстом) і використовується для створення змінних, доступних в усіх модулях програмного коду, де вони оголошені. Під час ініціалізуваннія змінної у тій же настанові, специфікатор `extern` ігнорується;

– `static` – статична змінна, яка має постійний час існування. Вона ініціалізується один раз при першому виконанні настанови, яка містить визначення значення змінної. Залежно від розташування настанови, описані статичні змінні можуть бути глобальними і локальними. Глобальні статичні змінні видимі тільки у тому модулі, у якому вони описані;

– `register` – аналогічний до специфікатора `auto`, але пам'ять виділяється за можливості в регістрах процесора і за відсутності такої можливості у компілятора змінні обробляються як `auto`.

Наприклад, фрагмент програмного коду:

```
int a;            // 1 глобальна змінна
int main(){
int b;            // 2 локальна змінна
extern int x;     // 3 змінна x визначена
                  // у іншому місці
static int c;     // 4 локальна статична
                  // змінна c
a=1;              // 5 присвоєння глобальній
                  // змінній
int a;            // 6 локальна змінна a
a=2;              // 7 присвоєння локальній
                  // змінній
::a=3;            // 8 присвоєння глобальній
                  // змінній
return 0;
}
int x=4;          // 9 визначення
                  // та ініціалізування x
```

У цьому прикладі глобальна змінна `a` визначена поза всіма блоками. Пам'ять для неї виділяється в сегменті даних на початку роботи програми, областю дії є вся програма.

Область видимості – вся програма, крім рядків 6–8, тому оскільки першому з них визначається локальна змінна з тим же ім'ям, область дії якої починається з початку її опису і закінчується при виході з блока. Змінні `b` і `c` – локальні, область їх видимості – блок, але час існування різний: пам'ять під `b` виділяється в стеку при вході у блок і звільняється при виході з нього, а змінна `c` розташована у сегменті даних та існує увесь час роботи програми.

Якщо початкове значення змінних явно не задається, компілятор присвоює глобальним і статичним змінним нульове значення відповідного типу. Автоматичні змінні не ініціалізуються.

Початкове ініціалізування змінних не є обов'язковим, проте все ж його бажано здійснювати.

Ім’я змінної є прикладом загальнішого поняття – *L*-вираз. Так називають вираз, який можна записати ліворуч (`left`) у настанові присвоєння. Він позначає змінну. Вираз, який позначає значення й записується праворуч (`right`) у виразах присвоювання, називається *R*-виразом (про це детально піде мова далі).

3.3.3. Вирази та оператори

У С++ компілятор також здатний обробляти вирази. Вираз (*expression*) – це математичний об’єкт, який створюється (складається) для проведення обчислень та отримання відповідного результату.

Проте термін "математичний об’єкт" дещо розмитий. Точніше буде так: вираз – це комбінація літералів, змінних, функцій та операторів, який генерує (створює) певне значення.

Наприклад, у математиці вираз 2 + 3 має значення 5. Вирази в мові С++ можуть містити :

– окремі цифри та числа (наприклад, 2, 45);
– літерні змінні (наприклад, `x`, `y`);
– оператори, зокрема, математичні (наприклад, +, -, *, /);
– функції.

Вирази можуть складатися і з окремих символів – цифр або літер (наприклад, 2 або `x`), і з різних комбінацій цих символів з операторами (наприклад, 2 + 3, 2 + `x`, `x` + `y` або (2 + `x`) * (`y` - 3)).

Літерали, змінні та функції ще отримали назву операнди. Операнди – це дані, з яких формується вираз за певними правилами. Літерали мають фіксовані значення, змінним можна надавати значення, функції ж генерують певні значення (залежно від типу повернення результату, винятком є функції типу `void`).

Останнім чинником у формуванні поняття вираз є оператори (користувачі, найімовірніше, звикли до терміна – операції). З їхньою допомогою ми можемо використати операнди для отримання нового значення.

Наприклад, у виразі 5 + 2 символ "+" є оператором. За допомогою оператора "+" ми об'єднали операнди 5 та 2 для отримання нового значення (7).

Оператори бувають трьох типів:

1. Унарні. Виконуються над одним операндом. Наприклад, оператор "-" (мінус). У виразі -7 оператор "-" застосовується тільки до одного операнда (7), щоб отримати нове значення (-7).

2. Бінарні. Виконуються із двома операндами (лівим та правим). Наприклад, оператор "+". У виразі 5 + 2 оператор "+" працює з лівим операндом (5) і правим (2), щоб створити нове значення (7).

3. Тернарні. Виконуються із трьома операндами (у мові C++ є лише один тернарний оператор).

Зауважте, що деякі оператори можуть мати кілька значень. Наприклад, оператор "-" (мінус) може використовуватися у двох контекстах: як унарний для зміни знака числа (наприклад, конвертувати 7 у -7 і навпаки), і як бінарний для виконання арифметичної операції віднімання (наприклад, 4 - 3).

Символ дорівнює "=" є оператором присвоєння. Деякі оператори складаються з більш ніж одного символу, наприклад, оператор рівності "==", який дозволяє порівнювати між собою два певні значення.

3.3.4. Ключові слова

Ключовими (службовими, зарезервованими) словами називають низку зарезервованих ідентифікаторів, які вживаються для формування конструкцій мови і мають фіксоване значення. За змістовим навантаженням службові слова поділяються на такі основні групи:

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

– кваліфікатори типів – `const` і `volatile`;

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

– для формування настанов – for, while, do, if, else, switch, case, continue, goto, break, return, default, sizeof.

У таблиці 3.3 подано перелік основних ключових слів С++.

Таблиця 3.3

Ключові слова С++

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

Як розділювачі лексем застосовуються такі символи: пропуск, табуляція, символ нового рядка, коментар. Між довільними двома лексемами допускається довільна кількість символів-розділювачів.

Крім того, деякі лексеми ("*","+",","" (","–>" тощо) самі є розділювачами і відділяти їх від інших лексем символами-розділювачами необов'язково.

3.3.5. Коментарі

Коментарі або починаються з двох символів ″скісна риска″ (//) і закінчуються символом переходу на новий рядок, або записуються між символами-дужками /* та */. Всередині коментаря можна використовувати довільні допустимі символи, а не тільки символи з алфавіту С++, оскільки компілятор коментарі ігнорує.

Коментарі необхідні для пояснень призначення тих або інших частин програмних кодів і їх текст завжди ігнорується компілятором. Мова С++ використовує два різновиди коментарів:

– *// текст* – однорядковий коментар, який починається з двох символів "/" і закінчується символом переходу на новий рядок;

– */* текст */* – багаторядковий коментар, що розташовується між символами–дужками "/*" і "*/".

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

Подамо кілька порад стосовно раціонального складання коментарів:

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

– пропуск – один з найбільш ефективних коментарів, що значно поліпшує розуміння програмного коду;

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

3.4. Концепція типів даних

Пам’ять комп’ютера складається з послідовностей бітів, а тому всі дані, які зберігаються в пам’яті, є двійковими кодами.

Тривалий час здавалося, що можна обходитися лише ними, даючи програмі змогу під час виконання "на власний розсуд"

динамічно інтерпретувати один і той самий код залежно від обставин. Однак такий спосіб безтипового програмування виявився надто небезпечним. Навіть у випадку числових величин, ґрунтуючись лише на двійковому коді, не завжди можна правильно відтворити значення об’єкта: адже один і той самий двійковий код може кодувати різні значення залежно від типу об’єкта. І навпаки: як побачимо далі, одне й те саме, навіть найпростіше числове значення, наприклад число 1, залежно від наданого йому типу, може бути подано різними двійковими кодами. Тому знання типу об’єкта – вирішальний чинник для визначення допустимих операцій над ним.

Про типи величин важливо домовитися на етапі проєктування, а не виконання програмного коду. Програми, у яких заздалегідь точно визначено типи всіх величин, надійніші, бо в них менше причин для непорозумінь. Мови програмування, які жорстко приписують кожному об’єкту програми його незмінний тип, як, наприклад, у *Pascal*, називають сильно типізованими (*strongly-typed*) на відміну від слабко типізованих (*weakly-typed*) мов, що дають змогу по-різному трактувати тип одного й того самого об’єкта.

Тяжіння до дедалі сильнішого типізування – загальна тенденція розвитку програмування загалом. Кожному об’єкту програми (і, відповідно, області пам’яті, яка зберігає його як елемент даних) присвоєно певний тип, від якого залежить інтерпретування даних, а саме спосіб їх кодування і декодування та, головне, набір допустимих операцій над ними.

Навіть якщо області даних різних типів перетинаються (наприклад, цілі та дійсні числа), тоді спосіб кодування числової величини залежить від присвоєного їй типу.

Зазначимо, що машинний код цілого числа 1 і дійсного числа 1, як уже було сказано, відрізняються.

Основна мета функціонування довільної програми полягає у обробленні початкових даних за певним алгоритмом. Дані різних типів зберігаються і обробляються по різному. У довільній алгоритмічній мові кожна константа, змінна, результат обчислення виразу або функції повинні мати визначений тип.

Усі дані характеризується класом пам’яті, ім’ям, типом і значенням. Імена дають змогу ідентифікувати дані, тобто відрізняти їх між собою. Програміст обирає тип кожного значення, який використовується для подання реальних об’єктів.

Тип задає множину допустимих значень даних і способи їх зберігання, перетворення та використання.

Тип даних визначає:

– внутрішнє подання даних у пам'яті комп'ютера;

– множину значень, які може приймати значення визначеного типу;

– операції і функції, які можна застосувати до значень визначеного типу.

Отже, тип даних – це множина значень разом із множиною операцій, які до них застосовуються. Множина значень називається носієм типу, множина операцій – сигнатурою.

Беручи до уваги викладені характеристики, програміст визначає тип кожного значення, яке використовується для подання реальних об'єктів.

Обов'язкове оголошення типу даних дозволяє компілятору робити перевірку допустимості різних конструкцій програми. Від типу значення залежать машинні команди, які будуть використовуватися для оброблення даних.

3.5. Базові (фундаментальні) типи даних

Для опису основних типів мови C++ використовують такі службові слова:

– `int` (цілий);
– `char` (символьний);
– `wchar_t` (розширений символьний)
– `bool` (логічний);
– `float` (дійсний);
– `double` (дійсний з подвійною точністю);
– `void` (без типу).

Почнемо з типу `void`, який лише формально віднесено до базових типів. Множина значень цього типу є порожньою, не має значень. Він використовується для визначення функцій, які не повертають значення, для визначення порожнього переліку аргументів функції, як основний тип для вказівників.

Усі подані фундаментальні типи, окрім `void`, є арифметичними в тому розумінні, що для кожного з них задано звичайні арифметичні операції. Логічний, символьний і цілі типи належать до так званих інтегральних типів *(integral types)*. Над ними, крім арифметичних, можна виконувати також логічні операції. Крім інтегральних типів, якими кодуються цілі числа, існують типи для зображення дійсних чисел. Це так звані дійсні типи з плаваючою крапкою *(floating point)*: `float`, `double` і `long double`, які відрізняються одне від одного точністю.

Типи `int`, `char`, `bool` називають *цілими*, а типи `float` та `double` – *дійсними з плаваючою крапкою*. Код, який формує компілятор для оброблення цілих значень, відрізняється від коду для значень з плаваючою крапкою.

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

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

У таблиці 3.4 подано діапазони значень та розміри базових типів даних (для 16-розрядного і 32-розрядного процесорів). Розмір однакового типу даних може відрізнятися на комп’ютерах різних платформ, а також може залежати від використаної операційної системи. Тому під час оголошення тієї або іншої змінної потрібно чітко уявляти, скільки байт вона буде займати в пам’яті комп’ютера, щоб запобігти проблемам, які пов’язані з переповненням і неправильним інтерпретуванням даних. Діапазони кожного з типів повинні бути перевірені для конкретного комп’ютера.

Символьне і булеве значення займають по одному байту. Зрозуміло, що однобайтові величини можуть набувати $2^8 = 256$ різних значень від 0 до 255.

Двобайтові величини можуть набути $2^{16} = 32\ 678$ значень із діапазону від 0 до 32 677.

Чотирибайтові величини можуть набути $2^{32} = 4\ 294\ 967\ 296$ значень із діапазону від 0 до 4 294 967 295. Зрозуміло, що чотирибайтових цілих типів не вистачить, наприклад, для точного підрахунку мешканців земної кулі.

Таблиця 3.4

Базові типи даних

Тип	Розмір, байт	Значення
`bool`	1	true або false
`unsigned short int`	2	від 0 до 65 535
`short int`	2	від -32 768 до 32 767
`unsigned long int`	4	від 0 до 4 294 967 295
`long int`	4	від -2 147 483 648 до 2 147 483 647
`Int (16 розрядів)`	2	від -32 768 до 32 767
`int (32 розряди)`	4	від -2 147 483 648 до 2 147 483 647
`unsigned int (16 розрядів)`	2	від 0 до 65 535
`unsigned int (32 розряди)`	4	від 0 до 4 294 967 295
`char`	1	від 0 до 256
`float`	4	від 1.2e-38 до 3.4e38
`double`	8	від 2.2e-308 до 1.8e308
`long double`	10	від 3.4e-4932 до 3.4e+4932

Зауважимо, що пам'ятати точне значення кожного степеня двійки не обов'язково, оскільки в багатьох випадках достатньо його наближення, обчисленого за допомогою так званої основної тотожності інформатики $2^{10} \approx 10^3$.

Тоді, наближено $2^{32} = 2^2 \cdot (2^{10})^3 \approx 4 \cdot (10^3)^3 = 4 \cdot 10^9 = 4\ 000\ 000\ 000$, що дає нам уявлення про порядок величини.

Поданим способом нескладно визначити порядок значень восьмибайтових величин

$$2^{64} = 2^4 \cdot 2^{60} = 16 \cdot (2^{10})^6 \approx 16 \cdot (10^3)^6 = 16 \cdot 10^{18} =$$
$$= 16\ 000\ 000\ 000\ 000\ 000\ 000.$$

Розділ 4

СТРУКТУРА С++-ПРОГРАМИ. ОПЕРАТОРИ МОВИ С++

4.1. Структура С++-програми

Формально програмний код мовою С++ складається з таких основних частин типової структури:

– директиви препроцесорного оброблення;
– оголошення (опис зовнішніх змінних та функцій);
– функції програми;
– головна функція – програми `main()`.

Отримання виконуваного коду програми мовою С++ здійснюється в декілька етапів: препроцесорне оброблення тексту програми, компілювання, редагування зв'язків (компонування) та виконання. Препроцесорне оброблення програми здійснюється за допомогою спеціальної програми-препроцесора CPP.EXE перед компілюванням програми.

Задачею препроцесора (препроцесор – це частина компілятора (програма), яка проводить попереднє оброблення програми) є перетворення початкового коду програми. На виході препроцесора отримується змінений код програми мовою С++. Препроцесор – це унікальна особливість С ++.

Для цього проводиться спеціальний етап, який називається попереднім обробленням. Попереднє оброблення проводиться перед процесом компілювання, а спеціальні функції попередньо обробляються. У результаті отримується розширена програма С ++, а потім вона передається компілятору.

Препроцесор одні дії виконує за замовчуванням, інші дії вказуються за допомогою спеціальних директив у коді програми.

Директиви препроцесора (*preprocessor directive*) – це команди препроцесора, вони завжди починаються символом #. Наприклад,

```
#include <ім'я файлу>
```

вказує препроцесору, що вміст заданого файлу необхідно обробити так, якби він знаходився у початковій програмі в тій точці, в якій знаходиться ця директива.

Оголошення є записом, який містить опис деяких об'єктів. У мові С++ оголошення є різновидом настанов і повинні завершуватися символом " ; ". Оголошення, як і настанови, виконуються в процесі роботи програми і результатом виконання оголошення, здебільшого, є створення імен об'єктів та самих об'єктів із відповідними властивостями.

Оголошення, яке не є частиною тіла довільної функції, є глобальним оголошенням. Об'єкт, створений таким оголошенням, також називається глобальним, він може бути і простим, і динамічним. Простором імен, створених глобальними оголошеннями, є програма загалом.

Одна з функцій повинна мати ім'я `main()`. Її ім'я `main` фіксоване в усіх програмних кодах мовою С++ і завжди записується однаково. Виконання програмного коду починається з першої настанови цієї функції.

Визначення функції має такий формат:

```
<Тип_результату> <ім'я>([параметри])
{
Настанови тіла функції
}
```

Зазвичай функції застосовуються для обчислення деякого значення, тому перед іменем функції оголошується його тип.

При створенні програми враховують такі основні вимоги:

– усі використані константи, змінні, функції та нестандартні типи повинні бути оголошеними (описаними) до їхнього першого використання і ці оголошення можна розміщати в довільному місці програмного коду;

– кожна настанова мови С++ закінчується символом ";";

– фігурні дужки (" { " та " } ") окреслюють блок настанов і все, що подано між такими дужками, синтаксично сприймається як одна настанова;

– вкладені блоки повинні мати відступ у 3–4 символи, водночас блоки одного рівня вкладеності потрібно вирівнювати за вертикаллю.

Приклад структури програми, яка містить функції `main`, `F1`, `F2`:

```
директиви препроцесора
оголошення
int main() {
     настанови головної функції
}
int F1(){
     настанови функції F1
{
int F2(){
     настанови функції F2
}
```

Загалом програма складається з декількох функцій, які не перетинаються (тобто "вкладення" однієї функції в іншу неприпустиме). Перед функціями і між ними можуть бути записані оголошення об'єктів даних і настанови препроцесорного оброблення. Функції користувача, які викликаються у головній функції `main()`, необхідно обов'язково описати до їх використання. Дуже часто першою подають програму, яка виводить на екран рядок-привітання.

```
#include <iostream>
using namespace std;
int main ()
{
  cout<<"Hello, World!";
  return 0;
}
```

У першому рядку записано директиву препроцесора. Директиви записують в окремих рядках і починають символом #. Слово

include (включити) означає, що препроцесор перед компіляцією програми має під'єднати до неї вміст спеціального файлу зі складу системи програмування, ім'я якого iostream записане в кутових дужках. У цьому файлі оголошено засоби введення/виведення (ім'я cout, оператор << і багато інших). Без під'єднання цього файлу ім'я cout буде невизначеним, і компілятор повідомить про цю помилку.

Існують два способи під'єднання директиви #include:

– #include <filename>;
– #include "filename".

Різниця між ними полягає в тому, де препроцесор буде шукати файли-ресурси. Якщо ім'я взяте в лапки, тоді препроцесор шукає його в робочому поточному каталозі, де є і основний файл компілювання. Такий запис зазвичай використовують для включення заголовних файлів, написаних користувачами. Якщо ж ім'я файлу записане у кутових дужках <…>, що використовуються для файлів стандартної бібліотеки, тоді пошук буде проводитись залежно від конкретного реалізування компілятора, за можливістю, в наперед визначених каталогах.

Файл iostream є одним із багатьох заголовних (*header*) файлів (або h-файлів), які входять до складу системи програмування, тобто є стандартними. У директивах імена стандартних заголовних файлів записуються в кутових дужках. Багато стандартних заголовних файлів мають порожнє розширення, для решти традиційно використовують розширення h.

У другому рядку розташовано настанову компілятору "використати простір імен std". Не пояснюючи на цьому етапі значення слів "простір імен", скажемо лише, що простір імен std є стандартним. У системі програмування мовою C++ у ньому описано всі бібліотечні засоби останнього покоління. Завдяки поданій настанові спрощується доступ до бібліотечних засобів (один з них, з ім'ям cout, використовується у поданому прикладі).

Поданий приклад програмного коду складається з однієї функції з ім'ям main. Запис

```
int main()
```

у третьому рядку – це заголовок функції. Дужки () після імені main указують, що це ім'я саме функції. Тип результату int перед

іменем функції є скороченням слова *integer* і означає, що функція має повертати ціле значення.

Функція з ім'ям `main` називається головною функцією. Вона має бути в кожній програмі, адже саме з неї починається виконання програми і, зазвичай, нею закінчується. Ім'я `main` не є зарезервованим, але використовувати його з іншим призначенням не варто. Вміст рядків 4–7 утворює тіло функції, що починається символом "`{`" і закінчується "`}`".

У тілі функції задано дії у вигляді послідовності настанов. Настанова в п'ятому рядку задає виведення на екран повідомлення `Hello, World!`. Воно візуалізується у вікні консолі, яке має відкритися на екрані під час її виконання та зникнути після завершення. Текстове повідомлення, яке виводиться на екран, записується в лапках `"text"`.

Виконанням настанови в шостому рядку функція має повернути значення 0 (нуль).

Отже, запустимо програму на виконання. Чи встигнемо ми щось побачити? Відповідь залежить від системи програмування, яка використовується. Деякі, але не всі, системи програмування дають змогу переглядати вікно програми після її завершення.

З іншого боку, виконання програми можна затримати за допомогою бібліотечної функції `system`, наприклад:

```
#include <iostream>
using namespace std;
int main()
{
cout<<"Hello, World!";
cout<<endl;
system("pause");
return 0;
}
```

Настановою

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

виконання програми призупиняється і у вікні консолі появляється повідомлення, що треба натиснути довільну клавішу. Після натискання довільної клавіші програма завершується.

Завдяки попередній настанові

```
cout<<endl;
```

повідомлення виводиться у новому рядку. Якби цієї настанови не було, повідомлення з'являлося б відразу після слів `Hello, World!`.

Пропонуємо з'ясувати, аналізуючи подані приклади, яка різниця у виконанні цих програмних кодів.

Приклад 4.1

```
#include <iostream>
using namespace std;
int main() {
cout<<"Microsoft";
cout<<"Visual";
cout<<"Studio";
cout<<endl;
system("pause");
return 0;
}
```

Приклад 4.2

```
#include <iostream>
using namespace std;
int main() {
cout<<"Microsoft ";
cout<<"Visual ";
cout<<"Studio";
cout<<endl;
system("pause");
return 0;
}
```

Приклад 4.3

```
#include <iostream>
using namespace std;
int main()
{
cout<<"Microsoft";
cout<<endl;
cout<<"Visual";
cout<<endl;
cout<<"Studio";
cout<<endl;
system("pause");
return 0;
}
```

4.2. Пріоритет виконання операторів і правила асоціативності

Для оброблення початкових даних мова C++ застосовує широкий набір операторів (див. табл. 4.1), що виконують формування і, відповідно, подальше обчислення виразів.

Оператор (*operator*), у літературних джерелах також знак операції – це позначення операції, виконання якої над числами та іншими значеннями породжує число або інше значення. Значення, до яких застосовується оператор, називаються операндами (*operand*), а породжуване значення – результатом (*result*).

Щоб правильно обчислювати вирази (наприклад, 4 + 2 * 3), ми повинні знати, що виконують певні оператори і в якій послідовності вони виконуються. Послідовність, в якій вони виконуються, називається *пріоритетом*.

У мові C++ всі оператори (операції) мають свій рівень пріоритету. Ті, в яких він вищий, виконуються першими. У таблиці 4.1 можна побачити, що пріоритет операторів множення і ділення (5) вищий, ніж в операторів додавання і віднімання (6). Компілятор

використовує пріоритет операторів для визначення порядку оброблення виразів.

Таблиця 4.1

Оператори мови С++

Асоціативність	Оператор	Опис	Приклад
1	2	3	4
1. Ні	::	Глобальна область видимості (унарний)	::ідентифікатор
	::	Область видимості класу (бінарний)	назва_класу:: назва_члену
2. L -> R	()	Круглі дужки	(вираз)
	()	Виклик функції	назва_функції (параметри)
	()	Ініціалізування	назва_типу(вираз)
	{}	uniform-ініціалізування (C++11)	назва_типу{вираз}
	type()	Конвертування типу	новий_тип(вираз)
	type{}	Конвертування типу (C++11)	новий_тип{вираз}
	[]	Індекс масиву	вказівник[вираз]
	.	Доступ до члена об’єкта	об’єкт.назва_члена
	->	Доступ до члена об’єкта через вказівник	вказівник_об’єкту ->назва_члена
	++	Пост-інкремент	lvalue++
	—	Пост-декремент	lvalue—
	typeid	Інформація про тип під час виконання	typeid(тип) або typeid(вираз)
	const_cast	Cast away const	const_cast(вираз)

Продовження таблиці 4.1

1	2	3	4
	dynamic_cast	Type-checked cast	dynamic_cast(вираз)
	reinterpret_cast	Конвертування одного типу в інший	reinterpret_cast(вираз)
	static_cast	Type-checked cast під час компіляції	static_cast(вираз)
3. R -> L	+	Унарний плюс	+вираз
	–	Унарний мінус	-вираз
	++	Пре (пост)-інкремент	++lvalue (lvalue++)
	—	Пре(пост)-декремент	—lvalue (lvalue—)
	!	Логічне НЕ (NOT)	!вираз
	~	Побітове НЕ (NOT)	~вираз
	(type)	C-style cast	(новий_тип)вираз
	sizeof	Розмір у байтах	sizeof(тип) або sizeof(вираз)
	&	Адреса	&lvalue
	*	Розіменування	*вираз
	new	Динамічне виділення пам'яті	new тип
	new[]	Динамічне виділення масиву	new тип[вираз]
	delete	Динамічне видалення пам'яті	delete вказівник
	delete[]	Динамічне видалення масиву	delete[] вказівник
4. L -> R	->*	Вибір члена через вказівник	вказівник_об'єкту ->*вказівник_на_член
	.*	Вибір члена об'єкта	об'єкт.*вказівник_на_член

Продовження таблиці 4.1

1	2	3	4
5. L -> R	*	Множення	вираз * вираз
	/	Ділення	вираз / вираз
	%	Залишок	вираз % вираз
6. L -> R	+	Додавання	вираз + вираз
	–	Віднімання	вираз – вираз
7. L -> R	<<	Побітовий зсув ліворуч	вираз << вираз
	>>	Побітовий зсув праворуч	вираз >> вираз
8. L -> R	<	Порівняння: менше	вираз < вираз
	<=	Порівняння: менше/дорівнює	вираз <= вираз
	>	Порівняння: більше	вираз > вираз
	>=	Порівняння: більше/дорівнює	вираз >= вираз
9. L -> R	==	Дорівнює	вираз == вираз
	!=	Не дорівнює	вираз != вираз
10. L -> R	&	Побітове І (AND)	вираз & вираз
11. L -> R	^	Побітове виключне АБО (XOR)	вираз ^ вираз
12. L -> R	|	Побітове АБО (OR)	вираз | вираз
13. L -> R	&&	Логічне І (AND)	вираз && вираз
14. L -> R	||	Логічне АБО (OR)	вираз || вираз

Продовження таблиці 4.1

1	2	3	4
15. R -> L	?:	Тернарний умовний оператор	вираз ? вираз : вираз
	=	Присвоєння	lvalue = вираз
	*=	Множення з присвоєнням	lvalue *= вираз
	/=	Ділення з присвоєнням	lvalue /= вираз
	%=	Ділення з остачею і з присвоєнням	lvalue %= вираз
	+=	Додавання з присвоєнням	lvalue += вираз
	-=	Віднімання з присвоєнням	lvalue -= вираз
	<<=	Присвоєння з побітовим зсувом вліво	lvalue <<= вираз
	>>=	Присвоєння з побітовим зсувом вправо	lvalue >>= вираз
	&=	Присвоєння з побітовою операцією I (AND)	lvalue &= вираз
	|=	Присвоєння з побітовою операцією АБО (OR)	lvalue |= вираз
	^=	Присвоєння з побітовою операцією "Виключне АБО" (XOR)	lvalue ^= вираз
16. R -> L	throw	Генерування винятку	throw вираз
17. L -> R	,	Оператор "Кома"	вираз, вираз

А що робити, якщо у двох операторів у виразі однаковий рівень пріоритету і розміщені вони поруч? Який оператор компілятор виконає першим? А тут уже компілятор буде використовувати *правила асоціативності*, які вказують напрямок виконання операторів: зліва праворуч або справа ліворуч.

Наприклад, у виразі 3 * 4 / 2 оператори множення і ділення мають однаковий рівень пріоритету (5), а асоціативність 5-го рівня відповідає виконанню операторів зліва праворуч, отож: (3 * 4) / 2 = 6.

Оператором є деяка дія, яка виконується над одним (*унарна*) або декількома (*бінарна, тернарна*) операндами і мають позначення (наприклад, оператор перевірки на рівність – позначення ″==″, оператор обчислення залишку від ділення цілих чисел – позначення ″%″ тощо).

Оператори:

– *унарні* або *одномісні* – &, *, -, +, ~, !, ++, --, sizeof;

– *бінарні* або *двомісні* – +, -, *, /, %, <<, >>, &, :, ^, <, >, <=, ==, >=, !=, &&, ||, =,*=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=, ., ->, ,, (), [];

– умовний *тернарний* або *тримісний* оператор – ?: .

Декілька приміток: 1 означає найвищий рівень пріоритету, а 17 – найнижчий. Оператори з більш високим рівнем пріоритету виконуються першими. L -> R означає зліва праворуч. R -> L означає справа ліворуч.

Ця таблиця призначена насамперед для того, щоб Ви могли у довільний момент повернутися до неї для розв’язання можливих проблем з пріоритетом або асоціативністю.

Розглянемо основні оператори. Для прикладу подамо пояснення до арифметичних операторів:

– + – додає величину R до L;

– - – віднімає R від L;

– - – унарний оператор зміни знака величини R;

– * – множення R і L;

– / – ділення L на R;

– % – залишок від ділення величини L на величину R (для цілих чисел).

Наприклад, якщо `int g = 12;`, тоді оператор `%` у настанові присвоєння

```
g = g % 9;
```

присвоїть результат: `g = 3;`

Оператор ділення має два режими. Якщо обидва операнди є цілими числами, тоді оператор виконує цілочисельне ділення, тобто довільний дріб відкидається і повертається ціле значення без заокруглення, наприклад, 7 / 4 = 1.

Якщо один або обидва операнди типу з плаваючою крапкою, тоді виконуватиметься ділення типу з плаваючою крапкою. Тут уже дріб наявний. Наприклад, 7.0 / 3 = 2.333, 7 / 3.0 = 2.333 або 7.0 / 3.0 = 2.333 генерують один і той же результат.

Спроби виконати ділення на 0 (або на 0.0) стануть причиною збою у вашій програмі, це правило *не потрібно забувати!*

Мова C++ підтримує такі оператори порівняння (оператори логічних відношень):

– < (менше);
– <= (менше або дорівнює);
– > (більше);
– >= (більше або дорівнює);
– == (дорівнює);
– != (не дорівнює).

Оператори порівняння є бінарними. Результатом виконання (аналізу) операторів є значення логічного типу – `true` або `false`.

У мові програмування C++ застосовуються три оператори логічних операцій:

– && – логічне "І" (кон’юнкція), бінарний оператор;
– || – логічне "АБО" (диз’юнкція), бінарний оператор;
– ! – логічне "НІ" (заперечення), унарний оператор.

Операндами логічних операторів є логічні дані. Результатом виконання логічного оператора є значення логічного типу – істина (`true`) і неістина (`false`).

Операндами також можуть бути числові дані: довільне ненульове значення – це істина, нуль – неістина (цей підхід використовується в мові програмування C, в якій немає логічного типу даних `bool`).

Мова С++ містить тернарний оператор `?:`, який називається оператором розгалуження, або умовним, і використовує умови. Оператор розгалуження має формат:

`вираз_1 ? вираз_2 : вираз_3`

За його виконання аналізується значення `вираз_1` (першого виразу) та зводиться до логічного типу. Якщо це значення `true`, тоді результатом виконання оператора розгалуження буде значення `вираз_2` (другого виразу), інакше – `вираз_3` (третього виразу).

Пріоритет оператора розгалуження нижчий, ніж у логічних операторів, але вищий, ніж у присвоєння.

4.3. Оператори інкременту та декременту

Операції інкременту (збільшення на 1) і декременту (зменшення на 1) значень змінних настільки поширені, що у них є власні оператори в мові С++. Крім того, кожен з цих операторів має дві форми (версії) застосування: префіксну і постфіксну (префікс і постфікс):

– оператор інкременту (++) збільшує значення операнду на одиницю;

– оператор декременту (--) зменшує значення операнду на одиницю.

Якщо оператор інкременту (декременту) розміщений перед ідентифікатором змінної, тоді говорять про префіксну форму запису інкременту (декременту). Якщо оператор інкременту (декременту) записаний після ідентифікатора змінної, тоді говорять про постфіксну форму запису.

З операторами інкременту/декременту у префіксній формі все просто. Значення змінної `x` спочатку збільшується/зменшується, а потім вже обчислюється, наприклад:

```
int x = 5;
int y = ++x; // x = 6 і 6 присвоюється змінній y
```

А ось з операторами інкременту/декременту у постфіксній формі трохи складніше. Компілятор створює тимчасову копію змінної x, збільшує або зменшує оригінальний x (НЕ копію), а потім повертає копію. Тільки після повернення копія x видаляється, наприклад:

```
int x = 5;
int y = x++;  // x = 6, але змінній y
              // присвоюється 5
```

Розглянемо поданий фрагмент коду детально. По-перше, компілятор створює тимчасову копію x, яка має таке ж значення, що і оригінал (5). Потім збільшується початковий x з 5 до 6. Після цього компілятор повертає тимчасову копію, значенням якої є 5, і присвоює її змінній y. Тільки після цього копія x знищується. Отже, у поданому прикладі ми отримаємо y = 5 і x = 6.

Ось ще один приклад, який демонструє різницю між префіксною і постфіксною формами:

```
#include <iostream>
using namespace std;
 int main()
{
int x = 5, y = 5;
cout << x << " " << y << endl;
cout << ++x << " " << --y << endl;
// префіксна форма
cout << x << " " << y << endl;
cout << x++ << " " << y-- << endl;
// постфіксна форма
cout << x << " " << y << endl;
return 0;
}
```

Результат виконання програмного коду:

```
5 5
```

```
6 4
6 4
6 4
7 3
```

У рядку 7 змінні `x` і `y` збільшуються/зменшуються на одиницю безпосередньо перед обробленням компілятором, так що відразу виводяться їх нові значення, а у рядку 9 тимчасові копії (`x = 6`, `y = 4`) скеровуються у потік виведення `cout`, а вже тільки потім оригінальні `x` і `y` збільшуються/зменшуються на одиницю. Саме тому зміни значень змінних після виконання операторів постфіксної форми не видно до наступного рядка.

Префіксна форма збільшує/зменшує значення змінних перед обробленням компілятором, а постфіксна форма – після оброблення компілятором.

Подамо ще один приклад, де форма оператора інкременту (декременту) впливає на порядок виконання операцій у виразах.

Результати виконання команд `a = 2; b = 3 * ++a;` будуть такими:
`a = 3, b = 3 * 3 = 9.`

Тут використано оператор інкременту у префіксній формі: спочатку збільшується значення змінної `a` на одиницю, а пізніше обчислюється вираз.

Розглянемо настанови

```
c = 5;
d = (c++) + 4;
```

Тут спершу обчислюється вираз (для `d`) з `c = 5`, а потім збільшується значення змінної `c` на одиницю.

Тобто `d = c + 4 = 5 + 4 = 9, c = c + 1 = 5 + 1 = 6` (це оператор інкременту у постфіксній формі).

Розділ 5

НАСТАНОВИ ПРИСВОЄННЯ. ВВЕДЕННЯ/ВИВЕДЕННЯ ДАНИХ

5.1. Настанови присвоєння

Мова С++ успадкувала від С кілька варіантів присвоєнь:

– просте присвоєння (*simple assignment*), яке позначають символом дорівнює "=";

– складене присвоєння (*compound assignment*) – присвоєння, суміщене з оператором, позначене двома символами, а саме оператором, за яким іде символ присвоєння, наприклад "+=", "%=", "/=".

5.1.1. Прості присвоєння

Надати значення змінній можна за допомогою настанови присвоєння, яка копіює значення, отримане у результаті аналізу (обчислення) деякого виразу, і присвоює його змінній. Програмно це можна реалізувати настановою присвоєння (*assignment statement, дослівний переклад – заява про призначення*) яка має такий формат запису:

```
<ім'я_змінної> = <вираз>;
```

Спочатку аналізується (обчислюється) `вираз` праворуч від символу присвоєння, а потім його значення присвоюється змінній,

ім’я якої вказано ліворуч. Найпростішими виразами є константи та імена змінних, складніші описано нижче.

Настанова присвоєння повертає як результат присвоєне значення. Завдяки цьому в С++ допускається присвоєння, які мають вигляд:

```
a = (b = c = 1) + 1;
```

Приклади:

1. Якщо є символьна змінна `char c;`, тоді настанова `c ='0';` присвоює їй значення ‘0’.

2. Якщо є ціла змінна `int i;`, тоді настанова `i=13;` присвоює їй значення 13.

Не варто плутати поняття "ініціалізування" та "присвоєння". Із причин, які стануть зрозумілими трохи пізніше, для позначення ініціалізування використовують також дужки.

Використання дужок для ініціалізування видається слушним, оскільки таке позначення менше нагадує присвоєння, ніж символ дорівнює, наприклад:

```
int i(1), j(1), k(1);
unsinged long counter(0);
double a(0.0), b(1.0);
double eps(0.000001);
```

Незважаючи на певну схожість, код ініціалізування

```
int i=1;
```

принципово відрізняється від коду визначення значення змінної з подальшим присвоєнням їй значення

```
int i; i=1;
```

У першому випадку одиниця записується до комірки пам’яті під час її призначення змінній, а у другому – ця комірка спочатку містить якесь значення, а лише потім до неї буде записано одини-

цю. Як "поводиться" неініціалізована змінна можна дізнатися, наприклад, у такий спосіб:

```
int i;
cout<<i<<endl;
```

Тому, вживання неініціалізованих змінних є потенційно небезпечним, оскільки є можливість скористатися змінною до того, як вона набуде значення.

5.1.2. Складені присвоєння

Настанови присвоєння дуже часто мають вигляд, наприклад,

```
a = a + b;
a = a * b;
```

тобто, збільшують поточне значення змінної `a` на величину `b`, множать на `b` тощо. Такі присвоєння можна реалізовувати за допомогою спеціальних настанов у скороченій формі, які називають складеними присвоєннями. На місці `b` можна записати довільний вираз відповідного типу. Значенням настанови присвоєння є нове значення змінної у його лівій частині.

Наприклад, складені настанови присвоєння:

- `a += 10;` відповідає настанові `a = a + 10;`
- `a -= 10;` відповідає настанові `a = a - 10;`
- `a *= 10;` відповідає настанові `a = a * 10;`
- `a /= 10;` відповідає настанові `a = a / 10;`
- `a %= 10;` відповідає настанові `a = a % 10.`

В описаний спосіб можна, наприклад, замість простого присвоєння `number=number%10;`

можна використати складене присвоєння

```
number%=10;
```

і замість

```
degree=degree+pow(2,n);
```

записати

```
degree+=pow(2,n);.
```

5.2. Правила узгодження типів

Сумісність типів за присвоєнням – це можливість присвоювати змінним одного типу значення іншого. Усі базові типи мови C++ сумісні за присвоєнням один з одним. У виразі присвоєння змінна отримує значення, перетворене до її типу.

Утворення значення одного типу за значенням іншого називається перетворенням типу. Під час перетворення цілого значення до дійсного типу відповідне число зображається в цьому типі. За цим правилом кожне значення типу `int` може бути зображене в типі `double`. При перетворенні дійсного значення до цілого типу у ньому відкидається дробова частина. Якщо ціла частина, що залишилася, не належить діапазону чисел типу `int`, тоді результат перетворення буде помилковим.

У настановах присвоєння діє правило перетворення типу виразу, записаного справа від символу " = ", до типу змінної, якій присвоюється значення даного виразу. Водночас:

– якщо тип змінної ліворуч від символу " = " старший за тип виразу праворуч від символу " = ", тоді тип виразу "підтягується" до типу змінної;

– якщо тип змінної ліворуч від символу " = " молодший за тип виразу праворуч від символу " = ", тоді значення змінної перетворюється (обрізається) до типу змінної. Наприклад,

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

```
c = 3.8;
b = a * c;  // b = 7 (відкидається дробова
            // частина)
```

Для явного перетворення типів новий тип записується в круглих дужках.

```
int a = 2, b;
float c = 3.8;
b = a * (int)c;
```

Тоді `b = 2 * 3 = 6.`

```
b = (int)(c * a);
```

У результаті аналізу вираз дорівнює 7.6, а для значень цілого типу дробова частина відкидається, тому `b` = 7.

Арифметичні перетворення типів застосовуються при узгодженні типів операндів в арифметичних і побітових операторах, а також в операторах порівняння. Вони ґрунтуються на "підтягуванні" операнда молодшого типу до старшого (цей процес ще називають просування типів – *type promotion*). Встановлено таку ієрархію типів:

`char` < `short` ≤ `int` ≤ `long` < `float` < `double` < `long double`

Для цілих типів додатково діє умова `signed тип < unsigned тип`.

5.3. Введення/виведення даних

Обов’язковим етапом розв’язання навіть найпростішої задачі на комп’ютері не обходиться без введення початкових даних та виведення проміжних і остаточних результатів. Введення даних – це передавання інформації ззовні в оперативну пам’ять (із зовнішнього носія); виведення даних – зворотний процес, коли дані після

оброблення передаються з оперативної пам’яті на зовнішній носій. Зовнішнім носієм може слугувати дисплей, друкуючий пристрій, жорсткий диск, флеш-пам’ять тощо. Передавання даних у програму та виведення результатів виконання програми є необхідним елементом програми.

У мові програмування C++ використовують концепцію поелементного введення/виведення даних. Виконуючи введення даних з клавіатури, комп’ютер тимчасово зупиняє виконання програмного коду і очікує на введення значення для змінної. У відповідь потрібно з клавіатури набрати деяку послідовність символів, яка зображує значення (ці символи візуалізуються на екрані). Введені символи запам’ятовуються у буфері та передаються функціям введення тільки після натиснення клавіші Enter.

Буфер – це область пам’яті для тимчасового зберігання даних. Максимальний обсяг буфера становить 128 символів (байтів). Завдяки наявності буфера можливе редагування даних під час їх введення.

Організація введення та виведення даних у різних мовах програмування виконується по різному: або відповідними настановами, або за допомогою стандартних підпрограм. Так, у мові C++ немає вмонтованих засобів введення/виведення даних. Для організації відповідних дій використовуються стандартні бібліотечні функції.

Бібліотеки C++ підтримують два основних способи введення/виведення:

- потокове введення/виведення;
- форматоване введення/виведення.

5.4. Поняття потоку

У мові C++ основним поняттям введення/виведення даних є `потік` – послідовність символів або інших даних. У програмі *потік є представником фізичного файлу* на зовнішньому носії даних (диску, клавіатурі або екрані монітора), а оператори обміну даними із файлом зображено як оператори відбору даних із потоку або дописування (долучення) їх до нього.

У програмуванні існує поняття *стандартних файлів* введення/виведення – зазвичай ними є клавіатура та екран. У C++-програмі їм відповідають *стандартний потік введення* з ім'ям `cin` і *стандартний потік виведення* з ім'ям `cout`. Отже, імена `cin` і `cout`, означені у файлі `iostream`, насправді позначають не клавіатуру та екран, а потоки, які відповідають цим пристроям у програмному коді. На початку виконання програмного коду обидва потоки `cin` і `cout` є порожні.

Із засобів оброблення потоків розглянемо лише оператори *введення* або *вставляння (insertion)* даних у потік >> і *виведення* <<, або *відбору* даних з потоку.

Є два способи надати змінній значення – ввести із зовнішнього носія або присвоїти. У найпростішому випадку настанова введення значення змінної має формат:

`cin >> ім'я_змінної;`

і реалізовує введення даних з клавіатури. Нагадаємо, ім'я `cin`, як і `cout`, оголошені у файлі `iostream`.

У настанові введення можна записати кілька імен змінних, кожне після відповідного оператора >>. За виконання такої настанови треба набрати з клавіатури відповідну кількість початкових значень, відокремивши їх одним або кількома порожніми символами.

Приклади:

1. Наприклад, є змінна `c` і виконуються настанови

```
char c;
cin >> c;
```

Щоб присвоїти змінній `c` значення `'A'`, треба натиснути послідовно на клавіші `A` та Enter. Якщо перед клавішею `A` кілька разів натиснути на клавіші Enter або Space, тоді результат буде такий самий.

2. Наприклад, змінні

```
double a, b, c;
```

є коефіцієнтами квадратного рівняння. Надати їм дійсні значення можна за допомогою трьох окремих настанов:

```
cin >> a;
cin >> b;
cin >> c;
```

Те ж саме буде зреалізовано і однією настановою:

```
cin >> a >> b >> c;
```

В обох ситуаціях з клавіатури слід набрати три значення, натискаючи між ними на клавішу пробілу, табуляції або Enter.

Виконання настанов закінчиться, коли після третьої константи буде натиснуто на Enter (перелік введення буде вичерпано).

Практично перед кожною настановою введення з клавіатури варто записати настанову виведення, яка запрошує до введення значень і вказує, скільки цих значень та яких типів.

Приклад. Надання значень дійсним змінним, які відповідають значенням коефіцієнтів квадратного рівняння, можна оформити таким фрагментом:

```
cout << "Ведіть значення коефіцієнтів квадратно-
го рівняння\n";
cout << "(Три значення дійсного або цілого
типу):\n";
cin >> a >> b >> c;
```

Після появи цього запрошення курсор буде переведено до початку наступного рядка екрана та почнеться відтворення символів, набраних з клавіатури.

Вивести значення змінної на екран можна за допомогою настанови відбору з потоку виведення, яка має формат:

cout << ім'я_змінної;

Настанова

cout << вираз;

обчислює та виводить результат обчислення виразу.

Наприклад, фрагмент програмного коду:

```
cout<< "Введіть число: ";
cin>> x;
cout<< "Квадрат цього числа: " << x*x << endl;
```

Приклад. Після введення значень трьох дійсних змінних за настановою

```
cin >> a >> b >> c;
```

можна вивести їх на екран, відокремивши пробілами:

```
cout << a;
cout << ' ';
cout << b;
cout << ' ';
cout << c;
```

Ті ж дії можна виконати й однією настановою:

```
cout << a << ' ' << b << ' ' << c;
```

Якщо з клавіатури введено значення 1, –3, 2, тоді буде надруковано `1 -3 2`.

Пропонуємо вам з'ясувати, що буде виведено на екран, якщо під час виконання поданого програмного коду введено символи `15` і `n` та натиснуто на клавішу Enter?

```
#include <iostream>
using namespace std;
int main()
{
int n; char a;
cin >> n >> a;
cout << "n=" << n<< ' ' << "a=" << a << endl;
system("pause");
return 0;
}
```

Зауважимо, що під час виведення тексту для коректного візуалізування літер кирилиці потрібно застосувати команди:

```
setlocale(0,".1251");
```

або

```
setlocale(LC_ALL, "Ukrainian");
```

Виведення замість літер кирилиці набору різних символів спричинено тим, що, наприклад, Visual Studio в консольних застосунках використовує для набраного тексту кодування Windows 1251, а для введеного тексту – кодування DOS. Коректно візуалізувати введений за допомогою `cin>>` текст надасть змогу команда

```
setlocale(LC_ALL, ".OCP");
```

повернувши початкові налаштування кодування.

Крім того, встановити потрібну кодову таблицю для потоків введення/виведення можна і такими способами:

```
system("chcp 1251");
system("chcp 1251 > null").
```

або

```
#include <Windows.h>
.........................................................
SetConsoleCP (1251);
SetConsoleOutputCP(1251);
```

Після цього у властивостях консольного вікна на вкладці Шрифт необхідно вибрати `Lucida Console`.

Доволі зручною є можливість виведення за допомогою `cout<<` чисел не лише в десятковому форматі, а і у шістнадцятковому або вісімковому, застосовуючи модифікатори `dec`, `hex` і `oct` усередині потоку виведення, наприклад:

```
#include <iostream>
using namespace std;
int main ()
{
setlocale(LC_ALL, "Ukrainian");
cout << "Вісімковий:\t\t " << oct << 10 << " "
<< 255 << endl;
cout << "Шістнадцятковий: \t " << hex << 10 << "
" << 255 << endl;
cout << "Десятковий:\t\t " << dec << 10 << " "
<< 255 << endl;
system("pause");
return 0;
}
```

Результатом виконання програмного коду будуть рядки:

```
Вісімковий: 12 377
Шістнадцятковий: a ff
Десятковий: 10 255
```

Варто зазначити, що використання одного з цих модифікаторів залишиться коректним, доки програма не завершиться, або не буде використано інший модифікатор.

5.5. Обчислення арифметичних виразів

Обчислення в арифметичних виразах виконуються зліва праворуч згідно з таким пріоритетом:

– стандартні функції, ++, --;
– множення (*), ділення (/), остача від ділення (%);
– додавання (+) та віднімання (–).

Спершу виконуються вирази, записані у круглих дужках. Для отримання правильного результату потрібно дотримуватися таких правил записування арифметичних виразів у настановах С++:

– кожна настанова має завершуватись крапкою з комою (`;`);
– мова С++ є чутливою до регістру, тобто `x` та `X` – це два різні ідентифікатори різних змінних;
– аргумент функції завжди записують у круглих дужках `ім’я_функції(аргументи);`
– знаки множення не можна пропускати (`3ab -> 3*a*b`);
– якщо знаменник або чисельник має оператори (+, −, *, /), тоді його слід записувати у круглих дужках;
– для записування раціональних дробів, у чисельнику або знаменнику яких є числові константи, хоча б одну з цих констант потрібно записати як дійсне число з обов’язковим записом десяткової крапки, наприклад, `2/k` записують як `2.0/k`;
– радикали (тобто корінь кубічний і вище) замінюють на дробові степені, наприклад, $\sqrt[3]{x+1}=(x+1)^{\frac{1}{3}}$ і записують як `pow(x+1, 1/3.0);`
– потрібно враховувати правила зведення типів, оскільки в арифметичних виразах можуть брати участь різнотипні дані та відбувається зведення типів.

У формуванні та аналізі арифметичних виразів використовують математичні оператори та математичні функції, подані у таблицях 5.1 та 5.2, відповідно.

Таблиця 5.1

Математичні оператори

Позначення	Оператор	Тип операндів і результату	Приклад
+	додавання	арифметичний, вказівник	x + y
-	віднімання та унарний мінус	арифметичний, вказівник	x - y
*	добуток	арифметичний	x * y
/	ділення	арифметичний	x / y
%	остача від ділення цілих чисел	цілий	i % 6
++	збільшення на одиницю (інкремент)	арифметичний, вказівник	i++; ++i
--	зменшення на одиницю (декремент)	арифметичний, вказівник	i--; --i

Таблиця 5.2

Математичні функції

<table>
<tr><th>Назва функції</th><th>Математичний запис</th><th>Назва функції</th><th>Математичний запис</th></tr>
<tr><td>cos(x)</td><td>cosx</td><td>fabs(x)</td><td>|x|</td></tr>
<tr><td>sin(x)</td><td>sinx</td><td>acos(x)</td><td>arccosx</td></tr>
<tr><td>tan(x)</td><td>tgx</td><td>asin(x)</td><td>arcsinx</td></tr>
<tr><td>log(x)</td><td>lnx</td><td>atan(x)</td><td>arctanx</td></tr>
<tr><td>log10(x)</td><td>lgx</td><td rowspan="2">ceil(x)</td><td rowspan="2">заокруглює число x до більшого цілого</td></tr>
<tr><td>pow(x,y)</td><td>x^y</td></tr>
<tr><td>sqrt(x)</td><td>$\sqrt[2]{x}$</td><td rowspan="2">floor(x)</td><td rowspan="2">відкидає дробову частину числа x</td></tr>
<tr><td>exp(x)</td><td>e^x</td></tr>
</table>

Розглянемо дві функції, означені в усіх реалізуваннях мови C++.

Функція `sqrt()` обчислює квадратний корінь свого невід'ємного дійсного аргументу. Значенням виразу `sqrt(2.0)` є приблизно 1.41421, а `sqrt(4.0)` – значення 2.0. *До значень цілого типу функція не застосовна*.

Функція `pow()` має два аргументи і обчислює піднесення до дійсного степеня, основою якого є перший аргумент, показником – другий. Наприклад, значенням виразу `pow(2.0, 3)` є 8.0, виразу `pow(2, 0.5)` – приблизно 1.41421. Результатом функції `pow()` *завжди* є дійсне значення.

Функція `log()` обчислює натуральний логарифм свого додатного дійсного аргументу, функція `log10()` – десятковий логарифм.

Застосування цих функцій до цілих аргументів є помилковим.

Функція `fabs()` обчислює дійсне значення |`x`| за дійсним аргументом `x`. Функція `abs()` із бібліотеки `cstdlib` обчислює ціле значення |`x`| за цілим аргументом `x`, якщо аргумент дійсний, обчислене значення може відрізнятися від математичного.

Приклади:

1. Корінь із невід'ємного дискримінанта квадратного рівняння з дійсними коефіцієнтами `a`, `b`, `c` можна обчислити виразом
`sqrt(b*b-4*a*c)`,
а дійсні корені рівняння – виразами
`(-b-sqrt(b*b-4*a*c))/(2*a)` та
`(-b+sqrt(b*b-4*a*c))/(2*a)`.

Дужки у знаменнику обов'язкові. Якщо їх не записати, тоді відбудеться не ділення, а множення на `a`.

2. Вираз `pow(b*b-4*a*c,0.5)` виконає обчислення квадратного кореня з `b*b-4*a*c`, вираз `pow(b,1.0/3.0)` – обчислення кубічного кореня з `b`, а обидва вирази `pow(2.0, 5)` та `pow(2, 5.0)` – піднесення дійсного числа 2.0 до степеня 5.

Зауважте: вираз `pow(2, 5)` із двома цілими аргументами є *помилковим*.

3. Значенням `log10(2.0)` є (наближено) 0.30103, значенням `log(1)` – дійсний (0.) нуль.

4. Значенням `fabs(-2.0)` є дійсне 2.0, значенням `abs(-2)` – ціле 2.

До речі, для обчислення значення логарифма за довільною основою доведеться використовувати вираз `log(x)/log(N)`.

У стандарті мови C++ відсутні математичні константи, зокрема ті, що позначають числа π = 3.141593… та e = 2.7182818….

Натомість у бібліотеці `cmath` означено константи з іменами `M_PI` – число π, `M_PI_2` – π /2, `M_PI_4` – π /4, `M_1_PI` – 1/ π, `M_E` (число e), `M_LN2` – ln 2, `M_LN10` – ln 10 і деякі інші. Щоб користуватися ними, необхідно перед під'єднанням бібліотеки `cmath` записати директиву

```
#define _USE_MATH_DEFINES
```

(*define* – означити).

Поданий приклад програмного коду виводить значення математичних констант π та e.

```
#include <iostream>
#define _USE_MATH_DEFINES
#include <cmath>
using namespace std;
int main()
{
cout << "pi=" << M_PI << endl;
cout << "e=" << M_E << endl;
cout << endl;
system("pause");
return 0;
}
```

Бібліотеки систем програмування мовою C++ містять різноманітні константи й низку підпрограм, які реалізовують матема-

тичні та інші функції. Зауважимо: склад бібліотек у різних середовищах може бути різним, тому вичерпну інформацію про вміст бібліотек може дати лише довідка в конкретному середовищі або самі бібліотечні файли.

Для того, щоб використати математичні функції, необхідно під'єднати відповідну бібліотеку:

```
#include <cmath>
```

Усі математичні функції приймають як аргументи і повертають числа з плаваючою крапкою подвійної точності `double`.

Приклад 5.1. Обмін місцями значень двох змінних. Алгоритм розв'язання цієї задачі детально описаний у першому розділі. Найпростіший спосіб обміняти дві змінні `a` та `b` місцями – використати проміжну змінну `c`.

```
#include <iostream>
using namespace std;
int main ()
{
// Встановлюємо локацію Україна
setlocale(LC_ALL, "Ukrainian");
// Оголошення змінних
double a, b, c;
// Ввід даних з консолі
cout << "Введіть значення a= ";
cin >> a;
cout << "Виведіть значення a= " << a << endl;
cout << "Введіть значення b= ";
cin >> b;
cout << "Виведіть значення b= " << b << endl;
c=a;
a=b;
b=c;
// вивід значень змінних a, b та c
// за результатами
```

```
// обміну місцями їх числових значень
cout << "\nВивід значень змінних після обміну їх місцями" << endl;
cout << "Значення a= " << a << endl;
cout << "Значення b= "<<b<<endl;
// Перевіримо значення змінної c
cout << "Значення c= " << c << endl;
system("pause");
return 0;
}
```

Приклад 5.2. Розробити проєкт програмного коду у С++ для обчислення

$$y = \frac{0.2x^2 - x}{\left(\sqrt{3} + x\right)(1 + 2x)} + \frac{2(x-1)^3}{sin^2 x + 1}$$

де *x* – довільна змінна, значення якої потрібно ввести з консолі.

```
#include <iostream>
#include <cmath>
using namespace std;
int main ()
{
setlocale(LC_ALL, "Ukrainian");
//Оголошення змінних
double y, x;
//Ввід даних з консолі
cout << "Введіть значення x= ";
cin >> x;
y=(0.2*x*x-x)/((sqrt(3.)+x)*(1+2*x)) +
2*pow(x-1,3.)/(pow(sin(x),2.)+1);
cout << "Результат y= " << y <<endl;
system("pause");
return 0;
}
```

Приклад 5.3. Розробити проєкт програмного коду у С++ для обчислення

$$y = \sqrt[3]{\frac{sin^2(mp - \pi)}{bx + p}} \ ;$$

$$x = arctg\left(b^2 + \sqrt{b + m}\right) \ i \ p = cos\left(e^{|b-x|} - lg|x - b|\right)^2.$$

Значення m = 7.1 задати як константу, а довільне значення b = 23.6 ввести з консолі.

```
#include <iostream>
#define _USE_MATH_DEFINES
#include <cmath>
using namespace std;
int main ()
{
setlocale(LC_ALL, "Ukrainian");
//Оголошення змінних і констант
const double m=7.4;
double b, x, p, y;
cout << "Введіть значення b= ";
cin >> b;
x = atan(b*b+sqrt(b+m));
p = cos(pow(exp(fabs(b-x))-log10(fabs
(x-b)),2));
y = pow(pow(sin(m*p-M_PI),2)/(b*x+p),1./3);
cout << "Результати:\n x= " << x << endl;
cout << " p= "<< p <<endl;
cout << " y= "<< y <<endl;
system ("pause");
return 0;
}
```

Результати роботи у консольному додатку:

```
Введіть значення b= 23.6
```

```
Результати:
x= 1.56902
p= 0.754801
y= 0.221877
```

Приклад 5.4. Розробити проєкт програмного коду у C++ для обчислення

$$f = \frac{e^{sinx} + \sqrt[4]{x+y}}{ln^2 zy}$$

Значення x=5.8, y=17.3, z=0.36 ввести з консолі.

```
#include <iostream>
#include <cmath>
using namespace std;
int main ()
{
//Встановлюємо локацію Україна
setlocale(LC_ALL, "Ukrainian");
//Оголошення змінних
float x,y,z;
float f;
//Вводимо з консолі значення початкових даних
cout << "\tx=";
cin >> x;
cout << "\ty=";
cin >> y;
cout << "\tz=";
cin >> z;
//Обчислюємо вираз
f = (exp(sin(y))+pow(x+y,1./4))/pow
(log(z*y),3.);
//Виводимо результат обчислення
cout  <<  "\nРезультат  обчислення:\n\tf(x,y,z)="
<< f << endl;
system ("pause");
return 0;
}
```

Розділ 6

СТВОРЕННЯ ПРОГРАМНИХ ПРОЄКТІВ РОЗГАЛУЖЕНОЇ СТРУКТУРИ У С++

У С++ застосовують низку настанов, які призначені для керування перебігом виконання програми. Є три категорії настанов керування:

– настанови вибору (`if`, `switch`);

– ітераційні настанови (які складаються з `for`-, `while`- і `do-while`-циклів);

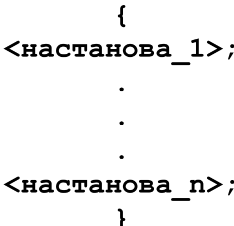
– настанови переходу (`break`, `continue`, `return` і `goto`).

6.1. Розгалужені алгоритми

Розгалуженням називається вибір програмою тієї, або іншої низки команд залежно від того, яким є результат аналізу певної умови (логічного виразу). Водночас виконується лише одна з гілок алгоритму.

Для програмного реалізування таких обчислень потрібно використовувати настанови керування перебігом виконання програми, котрі дають змогу змінювати порядок виконування настанов у програмі. У С++ для цього передбачено настанови: безумовного переходу – `goto`, умовного переходу – `if` та вибору варіанта – `switch`. Для формування умови переходу потрібно використовувати логічні (булеві) вирази.

Щоб написати кілька настанов там, де за правилами C++ має бути одна, наприклад, як гілку в настанові розгалуження, використовують блок настанов – послідовність настанов (дві і більше) у операторних дужках ″{ }″. Він має такий загальний формат:



```
{
<настанова_1>;
.
.
.
<настанова_n>;
}
```

Механізм програмного реалізування блоку настанов полягає у послідовному виконанні настанов, записаних у ньому (всередині операторних дужок "{ }").

6.2. Настанова умовного переходу if (настанова розгалуження)

Настанова if має дві форми запису: скорочену та повну.

Формат запису скороченої форми є таким:

```
if (<умова>) <настанова>;
```

Механізм реалізування настанови умовного переходу полягає в аналізі логічного виразу, який записаний в умові. Якщо, у результаті аналізу, умова набуває значення true, тоді виконуватиметься <настанова> (блок настанов). Якщо, у результаті аналізу, умова набуває значення false, тоді відбудеться перехід до виконання наступної після if настанови.

Фрагмент блок-схеми алгоритму, який ілюструє механізм функціонування настанови умовного переходу за скороченою формою запису подано на рисунку 6.1.

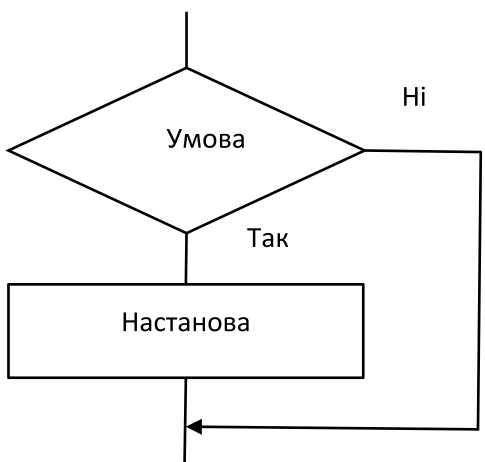


Рис. 6.1. Фрагмент блок-схеми алгоритму механізму функціонування настанови умовного переходу за скороченою формою запису

Подамо декілька прикладів використання скороченої форми настанови умовного переходу `if`.

Приклад 6.1. Задано три цілих числа a, b, c. Розробити проєкт програмного коду для обчислення мінімального значення серед цих чисел.

Проєкт програмного коду для розв’язання задачі:

```
#include <iostream>
using namespace std;
int main ()
{
    setlocale(LC_ALL, "Ukrainian");
int a, b, c;
int min; // мінімальне значення
cout << "\nВведіть значення a=";
cin >> a;
cout << "\nВведіть значення b=";
cin >> b;
```

```
cout << "\nВведіть значення c=";
cin >> c;
// обчислення мінімального значення серед цих
// чисел
min = a;
if (min > b) min = b;
if (min > c) min = c;
cout << "\nmin =" << min << endl;
system ("pause");
return 0;
}
```

Приклад 6.2. Задано ціле число n = 1, 2, 3, яке є номером обчислюваної функції. За значенням змінної n та x = 2.3 обчислити значення відповідної функції f(x):

1. $f(x) = x^2+8$;
2. $f(x) = -5x-1$;
3. $f(x) = 10-x$.

Проєкт програмного коду для розв'язання задачі з використанням скороченої форми настанови `if`:

```
#include <iostream>
using namespace std;
int main ()
{
    setlocale(LC_ALL, "Ukrainian");
int n;
float x, f;
// ввід значень n, x
cout << "Введіть значення n=";
cin >> n;
cout << "\nВведіть значення x=";
cin >> x;
if (n==1) f = x*x + 8;
if (n==2) f = -5*x - 1;
```

```
if (n==3) f = 10 - x;
cout <<"Результат f(x)= " << f << endl;
system ("pause");
return 0;
}
```

Формат повної форми запису настанови умовного переходу `if`:

```
if (<умова>) <настанова_1>;
     else <настанова_2>;
```

Механізм реалізування настанови умовного переходу за повною формою запису полягає в аналізі логічного виразу, який записаний в `умові`. Якщо, у результаті аналізу, `умова` приймає значення `true`, тоді виконуватиметься `<настанова_1>` (блок настанов), інакше – виконуватиметься `<настанова_2>` (блок настанов), після чого відбудеться перехід до виконання наступної після `if` настанови.

Зауважимо, що виконається лише одна з настанов, а не обидві.

Щоб програма краще сприймалася, настанову розгалуження часто записують у такому вигляді:

```
if (умова)
     <настанова_1>
else <настанова_2>
або
if (умова)
<настанова_1>
   else
   <настанова_2>
```

Фрагмент блок-схеми алгоритму, який ілюструє механізм функціонування настанови умовного переходу за повною формою запису подано на рисунку 6.2.

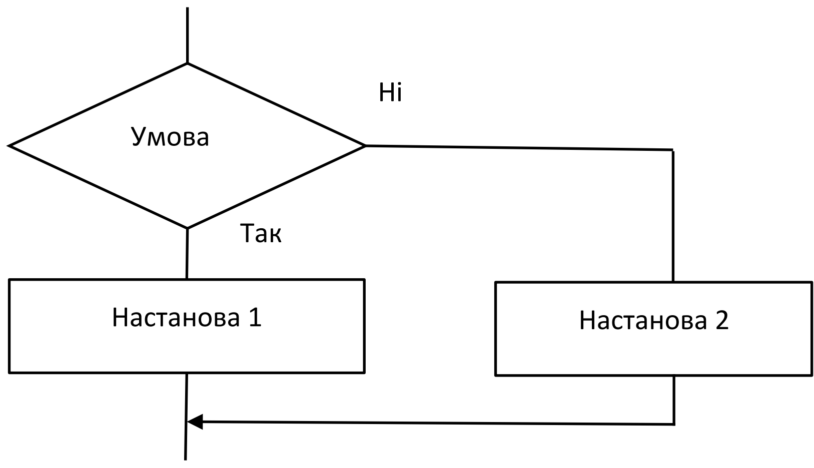


Рис. 6.2. Фрагмент блок-схеми алгоритму механізму функціонування настанови умовного переходу за повною формою запису

Наприклад, обчислення значення виразу y=f(x, b) де: y, x, b – змінні дійсного типу

$$y = \begin{cases} 1 + b^x & \text{за} \quad x = b \\ \dfrac{x + 3b}{b - x} & \text{за} \quad x \neq b \end{cases}$$

програмно можна реалізувати двома настановами `if` скороченої форми, або однією настановою `if` повної форми:

```
1. if(x == b) y = 1 + pow(b, x);
if(x != b) y = (x + 3*b)/(b - x);

2. if(x == b) y = 1 + pow(b, x);
else y = (x + b)/(b - x);
```

Ще один приклад. Заданий номер місяця року n. За номером місяця визначити, скільки днів у цьому місяці.

Фрагмент програмного коду для розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
    setlocale(LC_ALL, "Ukrainian");
int n;
int days;
// введення значення n
cout << "Введіть значення n= ";
cin >> n;
if ((n==4)||(n==6)||(n==9)||(n==11)) days = 30;
  else
if (n==2) days = 28;
  else
    days = 31;
cout << "\nРезультат: кількість днів " << days
<< " у " << n << " місяці" << endl;
system ("pause");
return 0;
}
```

Приклад 6.3. Перевести оцінку зі 100-бальної системи у 5-бальну за алгоритмом: оцінка менша за 50 відповідає незадовільній оцінці "2"; у межах від 50 до 74 – оцінці "3"; від 75 до 89 – "4"; починаючи з 90 і до 100 – "5".

Проєкт програмного коду для розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int m100, m5;
```

```
cout << "Введіть оцінку за 100-бальною шкалою: 
";
  cin >> m100;
if (m100 >= 90) m5 = 5;
  else
 if (m100 >= 75) m5 = 4;
   else
  if (m100 >= 50) m5 = 3;
    else m5 = 2;
cout << "Оцінка за 5-бальною шкалою: "<< m5 << 
endl;
system("pause");
return 0;
}
```

Приклад 6.4. За значеннями довжин трьох відрізків a, b, c, введених користувачем, визначити можливість існування трикутника, складеного з цих відрізків. Якщо такий трикутник існує, тоді визначити, чи є він різностороннім, рівнобедреним або рівностороннім.

Проєкт програмного коду для розв’язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
// Довжини сторін трикутника a, b, c
int a, b, c;
cout << "Введіть через пробіл значення 
a, b і c: ";
cin >> a >> b >> c;
  if (a + b <= c || a + c <= b || b + c <= a)
cout << "Трикутник не існує" << endl;
      else
  if (a != b && a != c && b != c)
cout << "Різносторонній" << endl;
```

```
else
  if (a == b && b == c)
cout << "Рівносторонній" << endl;
else
cout << "Рівнобедрений" << endl;
system("pause");
return 0;
}
```

Вкладені `if`-настанови.

Вкладені if-настанови формуються у тому разі, коли елементом `настанова` (див. повний формат запису) використовується інша `if`-настанова.

Вкладені `if`-настанови дуже популярні у програмуванні. Головне пам'ятати, що `else`-настанова завжди належить до найближчої `if`-настанови, яка знаходиться усередині того ж програмного блоку, але ще не пов'язана ні з якою іншою `else`-настановою.

Наприклад:

```
if(c)
{
if(d) <настанова1>;
if(f) <настанова2>; // Ця if-настанова
else <настанова3>;  // пов'язана з цією
                    // else-настановою.
}
else <настанова4>;  // Ця else-настанова
                    // пов'язана з if(c).
```

Як стверджується в коментарях, остання `else`-настанова не пов'язана з настановою `if(d)`, оскільки вони не знаходяться в одному блоці (незважаючи на те, що ця `if`-настанова – найближча, яка не має при собі "`else`-пари"). Внутрішня `else`-настанова пов'язана з настановою `if(f)`, оскільки вона – найближча і знаходиться усередині того ж блоку.

Дуже поширеною у програмуванні конструкцією, в основі якої знаходиться вкладена `if`-настанова, є "сходинки" `if-else-if`. Її можна подати у такому вигляді:

```
if(умова)
      <настанова1>;
else
      if(умова_1)
          <настанова2>;
      else
          if(умова_2)
              <настанова3>;
          else
                  <настанова4>;
```

У цьому записі під елементом `умова` розуміють логічний вираз, який обчислюється зверху вниз. Як тільки у довільній гілці виявиться істинний результат, тоді буде виконану настанову, пов'язану з цією гілкою, а всі решта "сходинки" ігноруються. Якщо виявиться, що жодна з умов не є істинною, тоді буде виконано останню `else`-настанову (можна вважати, що вона здійснює роль умови, яка діє за замовчуванням). Якщо останню `else`-настанову не задано, а всі інші виявилися помилковими, тоді взагалі ніяка настанова не буде виконана. Наприклад, визначити чи потрапляє значення `x` у діапазон від 1 до 4.

Проєкт програмного коду для розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int x;
cout << "Введіть значення x=";
cin >> x;
if(x==1) cout << "x дорівнює одиниці" << endl;
```

```
else if(x==2) cout << "x дорівнює двом" << endl;
else if(x==3) cout << "x дорівнює трьом" <<
endl;
else if(x==4) cout << "x дорівнює чотирьом" <<
endl;
else cout <<"x не потрапляє в діапазон від 1 до
4" <<endl;
system("pause");
return 0;
}
```

6.3. Приклади програмних проєктів, які реалізовують алгоритми з розгалуженою структурою у C++

Приклад 6.5.

Обчислити значення функції

$$y = \begin{cases} x^2 - 8 & \text{за} \quad x \le -4 \\ 3x - 2 & \text{за} \quad -4 < x < 0 \\ 2 - x & \text{за} \quad x \ge 0 \end{cases}$$

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
      setlocale(LC_ALL, "Ukrainian");
double y, x;
int n;
cout << "Введіть значення x=";
cin >> x;
if(x<=-4) { y=x*x-8; n=1;}
else
```

```
    if(-4<x&&x<0){ y=3*x-2; n=2;}
       else { y=2-x; n=3;}
cout << "\nРезультат y=" << y << endl;
cout << "\nВиконано умову " << n << endl;
system ("pause");
return 0;
}
```

Результат:

```
Введіть значення x=0.4
Результат y=1.6
Виконано умову 3
```

Обчислити значення функції

$$y = \begin{cases} \cos^{3.5}(a+zx) + e^{|bx|} & \text{за} \quad |1-x| = a+z \\ z + \ln|a+bx| & \text{за} \quad |1-x| > a+z \\ \sqrt{ab^4 + \sqrt[5]{zx^2}} & \text{за} \quad |1-x| < a+z \end{cases}$$

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
float a=0.3, b=1.25, x, y, z;
cout << "Введіть значення x= ";
cin >> x;
cout << "\nВведіть значення z= ";
cin >> z;
```

```
if(fabs(1-x*x)==(a+z))
y = pow(cos(a+x*z),3.5) + exp(fabs(b*x));
else if (fabs(1-x*x) > (a+z))
y = z+log(fabs(a+b*x));
else y = sqrt(a * pow(b,4.) + pow(z*x*x,1./5));
cout << "\nРезультат y = " << y <<endl;
system ("pause");
return 0;
}
```

Приклад 6.6. Необхідно обчислити одне із значень функції y = f(x) на інтервалі: $x \in [-5, 5]$ і номер умови, яка виконалася.

$$y = \begin{cases} \sin x^2 & \text{за} \quad |x| \le 1 \\ x^3 - 1 & \text{за} \quad -5 \le x < -3 \\ e^{x^2-3} & \text{за} \quad 3 \le x \le 5 \\ 12 & \text{у інших випадках} \end{cases}$$

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
cout <<"Програма для обчислення одного із зна-
чень функції на інтервалі -5;5\n" << endl;
//Оголошення змінних
int n;
float x, y;
//Ввід початкових даних з консолі
```

```
cout << "Введіть значення \tx=";
cin >> x;
//Аналіз значення аргументу функції
if ((x > 5) || (x < -5))
cout <<"\nЗначення аргументу за межами інтерва-
лу\n" << endl;
  else
{
if (fabs(x) <= 1)  //Перевірка першої умови
{
y = sin(x*x);
n=1;
cout << "Виконалася " << n << " умова\n" <<
endl;
cout << "\tЗначення функції y=" << y << endl;
}
  else if ((x >= -5) && (x < -3))  //Перевірка
// другої умови
{
y = pow(x, 3) - 1;
n=2;
cout << "\nВиконалася "<<n<< " умова\n" << endl;
cout << "\tЗначення функції y=" << y << endl;
}
  else if ((x >= 3) && (x <= 5)) //Перевірка
// третьої умови
{
y = exp(x*x - 3);
n=3;
cout << "\nВиконалася " << n << " умова\n" <<
endl;
cout << "\tЗначення функції y=" << y << endl;
}
      else        //В інших випадках
{
```

```
y = 12;
n=4;
cout << "\nВиконалася " << n << " умова\n" <<
endl;
cout << "\tЗначення функції y=" << y << endl;
}
}
system ("pause");
return 0;
}
```

6.4. Настанова безумовного переходу goto

Настанова `goto` (перейти до) дає змогу передавати керування у довільну точку програмного коду (програми), котру позначено спеціальною позначкою.

Формат настанови `goto`:

```
goto <позначка>;
```

Позначку записують перед настановою, якій необхідно передати керування перебігом виконання програми і відокремлюють від неї символом двокрапки (`:`). Позначки у С++ не оголошують. Як позначка може застосовуватись поєднання довільних літер латиниці та цифр, але розпочинатися позначка повинна з літери, наприклад: `start`, `M1`, `second`.

Застосовувати цю настанову потрібно обережно і помірковано, особливо під час переходу всередину блоку або циклу, оскільки це може призвести до непередбачуваних помилок. Тому, в С++ настанова `goto` застосовується вкрай рідко і вважається застарілою. Однак на практиці часто трапляються випадки, коли ця настанова значно спрощує код програми.

Наприклад:

```
<настанова_1>;
goto <позначка 1>;
<настанова_2>;
.
.
.
<настанова_n>;
<позначка 1>:
<настанова_n+1>;
```

Або

```
<настанова_1>;
<позначка 2>:
<настанова_2>;
.
.
.
<настанова_n>;
goto <позначка 2>;
<настанова_n+1>;
```

6.5. Настанови вибору варіантів `switch`

Настанова вибору варіантів `switch` дає змогу вибрати один варіант перебігу розв'язання задачі з декількох залежно від значення виразу-селектора. Так забезпечується багатонаправлене розгалуження.

Настанова `switch` може бути заміненою настановою `if`. Однак у деяких випадках використання настанови `switch` може бути більш ефективним, ніж використання настанови `if`.

Формат настанови switch:

```
switch (<вираз_селектор>)
{case <значення_позначка_1> : {<послідов-
ність_настанов>; break;}
.
.
.
case <значення_позначка_n> : {<послідов-
ність_настанов>; break;}
[ default: <послідовність_настанов>; ]
}
```

Слова switch, break, case, default є зарезервованими; вони позначають відповідно "перемикач", "випадок", "за відсутності".

Спочатку обчислюється вираз_селектор у дужках (селектор варіантів). Вираз_селектор повинен мати цілий або символьний тип. Значення виразу_селектора порівнюється зі значеннями_позначок (позначок варіантів) після ключових слів case. Якщо значення виразу_селектора збіглося зі значенням_позначки, тоді виконується відповідна послідовність_настанов, відзначена цим значенням_позначкою і записана після двокрапки, доки не зустрінеться настанова break.

Якщо значення виразу_селектора не збіглося з жодним значенням_позначкою, тоді виконуються настанови, які записані за ключовим словом default. (default є необов'язковою конструкцією настанови switch, на що вказують квадратні дужки [] у форматі запису.)

Настанова break здійснює вихід із switch. Якщо настанова break відсутня наприкінці настанов відповідного case, тоді буде почергово виконано всі настанови до наступного break, або до кінця switch для всіх гілок case, незалежно від значення їхніх значень_позначок.

Наприклад, за значенням введеного цілого числа n вивести назву дня тижня.

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int n;
cout << "Введіть ціле число від 1 до 7: ";
cin >> n;
switch (n)
{
case 1: cout<<"понеділок"; break;
case 2: cout<<"вівторок"; break;
case 3: cout<<"середа"; break;
case 4: cout<<"четвер"; break;
case 5: cout<<"п’ятниця"; break;
case 6: cout<<"субота"; break;
case 7: cout<<"неділя"; break;
default:  cout << "\nПомилка!
Число не є значенням від 1 до 7\n";
}
system("pause");
return 0;
}
```

6.6. Приклади використання настанови вибору switch

Приклад 6.7. За відомим значенням n = 1..7, що є номером дня тижня, визначити, вихідний цей день чи будній. Результат записати у змінну `fDayOff` типу `bool`.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
      setlocale(LC_ALL, "Ukrainian");
int day;
bool fDayOff;
cout << "\nВведіть номер дня тижня "<<endl;
cin >> day;
switch (day)
{
    case 1:
        fDayOff = false;
    break;
    case 2:
        fDayOff = false;
    break;
    case 3:
        fDayOff = false;
    break;
    case 4:
        fDayOff = false;
    break;
    case 5:
        fDayOff = false;
    break;
    case 6:
        fDayOff = true;
    break;
    case 7:
        fDayOff = true;
    break;
}
```

```
cout << "\nНомер дня тижня " << day << endl;
if(fDayOff) cout << "\nВихідний день" << endl;
else cout << "\nБудній день" << endl;
system("pause");
return 0;
}
```

Інший, більш компактний варіант програмного коду розв’язання цієї задачі:

```
#include <iostream>
using namespace std;
int main()
{
      setlocale(LC_ALL, "Ukrainian");
int day;
bool fDayOff;
cout << "\nВведіть номер дня тижня "<<endl;
cin >> day;
switch (day)
{
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
        fDayOff = false;
    break;
    case 6:
    case 7:
        fDayOff = true;
    break;
}
```

```
cout << "\nНомер дня тижня " << day << endl;
if(fDayOff) cout << "\nВихідний день" << endl;
else cout << "\nБудній день" << endl;
system("pause");
return 0;
}
```

Ще один варіант проєкту програмного коду розв’язання цієї задачі:

```
#include <iostream>
using namespace std;
int main()
{
      setlocale(LC_ALL, "Ukrainian");
int day;
bool fDayOff;
cout << "\nВведіть номер дня тижня " << endl;
cin >> day;
switch (day)
{
    case 6:
    case 7:
        fDayOff = true;
    break;
    default:
        fDayOff = false;
}
cout << "\nНомер дня тижня " << day << endl;
if(fDayOff) cout << "\nВихідний день" << endl;
else cout << "\nБудній день" << endl;
system("pause");
return 0;
}
```

Приклад 6.8. Задано ціле число n = 1..3, яке є номером функції. За значенням змінної n обчислити значення відповідної функції y = f(x):

1. $-2x^2-4$;
2. 5x+2;
3. 15-3x.

Проєкт програмного коду для розв’язання задачі з використанням настанови `switch`:

```
#include <iostream>
using namespace std;
int main()
{
      setlocale(LC_ALL, "Ukrainian");
// оголошення змінних
int n;
float f, x;
cout << "\nВведіть номер функції n= ";
cin >> n;
cout << "\nВведіть значення аргументу функції x=
";
cin >> x;
switch (n)
{
    case 1:
        f = -2*x*x-4;
    break;
    case 2:
        f = 5*x+2;
    break;
    case 3:
        f = 15-3*x;
    break;
}
```

```
cout << "\nНомер функції n=" << n << endl;
cout << "\nЗначення функції f=" << f << endl;
system("pause");
return 0;
}
```

Приклад 6.9. За введеною датою визначити день тижня, на який ця дата припадає. Для обчислення скористайтеся формулою

$$\left(d + \left[\frac{13m - 1}{5}\right] + y + \left[\frac{y}{4}\right] + \left[\frac{c}{4}\right] - 2c + 777\right) mod 7\,,$$

де: d – день місяця; m – номер місяця, починаючи рахунок від березня, як це робили у Стародавньому Римі (березень – 1, квітень – 2, … , лютий – 12); y – рік у столітті; c – кількість століть.

Квадратні дужки у формулі означають, що треба взяти лише цілу частину від значення у дужках. Обчислене за формулою значення визначає день тижня: 1 – понеділок, 2 – вівторок, … , 6 – субота, 0 – неділя.

Проєкт програмного коду для розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int day, month, year; //день, місяць, рік
int c, y, m; //століття і рік у столітті, місяць
             // за римським календарем
int weekday; //день тижня
cout << "Введіть дату: день місяць рік" << endl;
cin >> day >> month >> year;
if (month == 1 || month == 2) year--;
//січ. і лют. належать до мин.року
m = month - 2; // рік розпочинається з березня
if (m <= 0) m += 12; // для січня і лютого
c = year / 100;
```

```
y = year - c * 100;
weekday = (day + (13 * m - 1) / 5 + y + y / 4 +
c / 4 - 2 * c + 777) % 7;
switch (weekday)
{
case 1: cout << "Понеділок" << endl; break;
case 2: cout << "Вівторок" << endl; break;
case 3: cout << "Середа" << endl; break;
case 4: cout << "Четвер" << endl; break;
case 5: cout << "П’ятниця" << endl; break;
case 6: cout << "Субота" << endl; break;
case 0: cout << "Неділя" << endl;
}
system("pause");
return 0;
}
```

Важливі моменти використання настанови `switch`:

– настанова `switch` відрізняється від `if` тим, що вона може виконувати тільки операції перевірки строгої рівності, а `if` може аналізувати логічні вирази;

– не може бути двох констант в одній настанові `switch`, які мають однакові значення. Звичайно, настанова `switch`, яка охоплює іншу настанову `switch`, може містити аналогічні константи;

– якщо в настанові `switch` використовуються символьні константи, тоді вони автоматично перетворюються в цілочисельні.

Розділ
7

СТВОРЕННЯ ПРОГРАМНИХ ПРОЄКТІВ ЦИКЛІЧНОЇ СТРУКТУРИ У С++

Обчислювальний процес називається *циклічним*, якщо однотипні обчислення повторюються, доки виконується певна задана умова. Блок повторюваних настанов називають *тілом (областю дії)* циклу.

У С++ наявні три настанови циклу:

– настанова циклу `for`;
– настанова циклу з передумовою `while`;
– настанова циклу з постумовою `do-while`.

7.1. Настанова циклу **for**

У мові С++ настанова циклу `for` може мати дуже широке реалізування та застосування для циклічних обчислювальних процесів з відомою кількістю повторень однотипних обчислень. Цикл `for` ще називається циклом з параметром або параметричною настановою циклу.

Формат настанови:

```
for (<ініціалізування>; <умова>; <модифікування>)
                  <тіло циклу>;
```

Конструктивно ця настанова складається з трьох основних блоків, розміщених у круглих дужках після зарезервованого слова `for` і відокремлених один від одного крапкою з комою (;), та настанов (тіла циклу), які виконуються у цьому циклі певну кількість разів.

На початку виконання параметричної настанови циклу `for` *одноразово* у блоці `ініціалізування` задаються початкові значення керуючих змінних (параметрів), які керують циклом. Потім аналізується `умова` і, якщо у результаті аналізу вона набуде значення `true`, тоді виконується `тіло циклу` (одна, або блок настанов).

`Модифікування` змінює значення параметру циклу й аналізується `умова`.

Якщо у результаті аналізу вона набуде значення `true`, тоді виконання циклу триває.

Якщо `умова` не виконується (у результаті аналізу вона набуває значення `false`), тоді відбувається вихід із циклу і керування перебігом виконання програми передається настанові, яка записана за настановою `for`.

Вагомим є те, що аналіз `умови` виконується на початку циклу. Це означає, що тіло циклу може не виконатись жодного разу, якщо `умова` на першому кроці виконання циклу набуває значення `false`.

Кожне повторення (крок) циклу називається *ітерацією*.

Зауважте!

Якщо блок `умова` залишити порожнім, тоді результат аналізу `умови` завжди буде вважатися `true` і ми отримаємо "безконечний цикл".

Змінні, визначені всередині циклу `for`, мають спеціальний тип області видимості, область видимості циклу.

Такі змінні існують лише всередині циклу та недоступні за його межами.

Структура параметричної настанови циклу подана на рисунку 7.1.

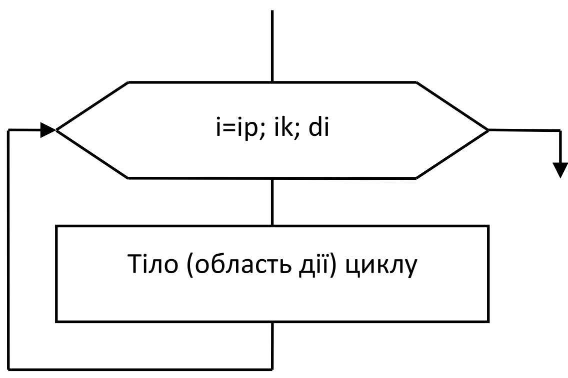


Рис. 7.1. Структура параметричної настанови циклу `for`

На цьому рисунку позначено ip, ik, di – початкове, кінцеве, крок зміни значення параметру циклу `i`, відповідно.

Наприклад, поданий далі програмний код за допомогою циклу `for` виводить на екран числа від 1 до 100.

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int vne;
for(vne=1; vne<=100; vne=vne+1)
cout << vne << " ";
system("pause");
return 0;
}
```

Спочатку змінна `vne` ініціалізується числом 1. Під час кожного повторення циклу аналізується умова `vne` < = 100. Якщо результат аналізу виявляється істинним, `cout`-настанова виводить значення змінної `vne`, після чого її вміст збільшується на одиницю.

Коли значення змінної vne перевищить значення 100, тоді умова, яка аналізується, набуває значення false і виконання циклу припиниться.

У професійно написаному С++-коді програми рідко трапляються настанови vne = vne + 1, оскільки для таких настанов у мові програмування С++ передбачений оператор *інкременту.*

За використання опрератора інкременту у попередньому програмному коді настанову for можна переписати так:

```
for(vne=1; vne<=100; vne++)
cout<<vne<<" ";
```

Розглянемо схожий приклад використання настанови циклу for і виконаємо аналіз виконання програмного коду. Необхідно вивести на екран числа від 0 до 9.

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

Спочатку, у блоці ініціалізування, ми оголошуємо змінну vne та присвоюємо їй початкове значення 0. Далі перевіряється умова vne < 10 (vne дорівнює 0), тоді, у результаті аналізу, умова 0 < 10 набуває значення true.

Отже, виконується тіло циклу, в якому здійснюється виведення у консольному додатку значення змінної vne (тобто значення 0).

Потім у блоці модифікування виконується оператор ++vne. Цикл знову повертається до аналізу умови. Умова 1 < 10 у результаті аналізу набуває значення true, тому тіло циклу виконується знову. Виводиться 1, а змінна vne збільшується вже

до значення 2. Умова 2 < 10 є істинною, тому виводиться 2, а `vne` збільшується до 3 і так далі.

Зрештою `vne` збільшується до 10, а умова 10 < 10 у результаті аналізу приймає значення `false` і цикл завершується.

Отже, результатом виконання програмного коду буде:

```
0 1 2 3 4 5 6 7 8 9
```

Хоча в більшості циклів у блоці `модифікування` використовується оператор інкременту параметра циклу, ми також можемо використовувати оператор декременту параматра циклу, наприклад:

```
#include <iostream>
using namespace std;
int main()
{
for (int vne = 9; vne >= 0; --vne)
cout << vne << " ";
return 0;
}
```

Результатом виконання програмного коду буде:

```
9 8 7 6 5 4 3 2 1 0
```

Також з кожною новою ітерацією ми можемо збільшити або зменшити значення параметра циклу більше ніж на одиницю:

```
#include <iostream>
using namespace std;
int main()
{
for (int vne = 9; vne >= 0; vne -= 2)
cout << vne << " ";
return 0;
}
```

Результатом виконання програмного коду буде:

```
9 7 5 3 1
```

Приклад. Обчислення суми $S = \sum_{i=1}^{10} i$ проілюструє використання настанови `for`.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
int s = 0;
for(int i = 1; i <= 10; i++) s += i;
cout << "\ns= " << s;
system("pause");
return 0;
}
```

Цю настанову `for` можна прочитати так: "виконати настанову `s + = i` 10 разів (для значень `i` від 1 до 10 включно, де `i` на кожній ітерації збільшується на 1) ". У поданому прикладі є два присвоєння початкових значень: `s = 0` і `i = 1`, умова продовження циклу: (`i< = 10`) і змінення значення параметра: `i++`.

Тілом циклу є настанова `s + = i`.

Порядок виконання цього циклу є такий:

– присвоюються початкові значення (`s = 0, i = 1`);

– аналізується умова (`i < = 10`);

– якщо результат аналізу умови є `true`) виконується настанова тіла циклу: до суми, обчисленої на попередній ітерації, додається нове число;

– параметр циклу збільшується на 1.

Тепер внесемо незначні зміни до попереднього прикладу:

```
int s = 0;
for(int i = 1; i <= 10; ++i) s += i;
```

Як, на Вашу думку, вплинуть внесені зміни на результат розв'язання задачі?

У настанові `for` можливі конструкції, коли є відсутній той або інший блок:

– `ініціалізування` може бути відсутнім, якщо початкове значення задати попередньо;

– `умова` – якщо припускається, що умова завжди набуває значення frue, тобто потрібно неодмінно виконувати тіло циклу, доки не трапиться настанова `break`;

– `модифікування` – якщо приріст параметра здійснювати в тілі циклу або взагалі це є непотрібним.

У згаданих випадках сам вміст блоку пропускається, але крапка з комою (;) неодмінно має залишитись. Можливою є наявність *порожньої* настанови (настанова є відсутньою) у тілі циклу. Наприклад, суму з попереднього прикладу можна обчислити іншим способом:

```
#include <iostream>
using namespace std;
int main()
{
int i, s;
for(i = 1, s=0; i <= 10; s += i++) ;
cout << "\ns= " << s;
system("pause");
return 0;
}
```

У цьому прикладі `настанова` відсутня. Початкові значення параметрові циклу `i` та змінній `s` присвоюються у заголовку циклу,

тому вони оголошені поза циклом. Наголосимо, що змінна `s` оголошена поза межами дії настанови циклу і таке оголошення `s` означає, що значення цієї змінної є визначеним у межах цілого програмного коду. Якщо б її оголошення було у розділі `ініціалізування`, тоді після завершення циклу значення `s` буде мати випадкове значення (невизначене) і задокументувати його неможливо.

Розглянемо використання циклу `for` для обчислення факторіалу F = n! (нагадаємо, що факторіал обчислюється за формулою n! = 1·2·3...(n–2) · (n–1) · n, наприклад: 4! = 1·2·3·4 = 24). Подамо три аналогічні за функціональним призначенням фрагменти форми запису настанови `for`:

```
1. int F=1, n=5;
   for(int i=1; i<=n; i++) F *= i;
2. int F, i, n=5;
   for(F=1, i=1; i<=n; F *= i++);
3. int F=1, i=1, n=5;
   for( ; i<=n; ) F *= i++;
```

Приклад 7.1. Вивести на екран результати піднесення числа 2 до степенів від 0 до N (N < 100).

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int n;
cout << "Ввести N<100 " << endl;
cin >> n;
for (int i = 0; i <= n; i++)
{
```

```
cout << "2^" << i << "=" << pow(2,i) << endl;
}
system("pause");
return 0;
}
```

Приклад 7.2. Вивести на екран квадрати цілих парних чисел від 10 до N включно.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int n;
cout << "Ввести N "<< endl;
cin >> n;
for (int i = 10; i <= n; i+=2)
cout << i << "^2 = " << i*i << endl;
system("pause");
return 0;
}
```

Приклад 7.3. Обчислення суми ряду. Ввести ціле число *n* і дійсне число *x*, обчислити суму *n* членів ряду

$$S = \frac{x}{4} - \frac{x^3}{6} + \frac{x^5}{8} - \cdots = \sum_{i=0}^{n} \frac{(-1)^{i+1} x^{2i-1}}{2(i+1)}$$

Проєкт програмного коду розв'язання задачі:

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

```
setlocale(LC_ALL, "Ukrainian");
double x, u, s=0;
int i, n;
cout << "Введіть ціле значення n= ";
cin >> n;
cout << "\nВведіть значення x= ";
cin >> x;
cout << "\nРезультати:\n";
for(i=1; i<=n; i++)
{
u = pow(-1,i+1)*pow(x,2*i-1)/(2*(i+1));
cout << "\nДоданок " << i << " = " << u << endl;
s+=u;
}
cout << "Сума = " << s << endl;
system("pause");
return 0;
}
```

У поданому прикладі змінна `i`, яка є параметром циклу, оголошена за межами області дії циклу, а це означає, що значення цієї змінної є визначеним у межах цілого програмного коду.

7.2. Впорядкування (опрацювання) послідовності чисел

Існує низка задач, в яких необхідно певним способом опрацьовувати задану числову послідовність, зокрема, для обчислення результату достатньо переглянути послідовність один раз. Наприклад, щоб обчислити середнє арифметичне заданої послідовності чисел, можна обчислення суми чисел і їхньої кількості поєднати з введенням чисел. Тоді не потрібно буде зберігати всю послідовність у пам’яті комп’ютера (у вигляді масиву), достатньо мати одну скалярну змінну цілого або дійсного типу та почергово присвоювати їй значення, які вводяться.

Числова послідовність може задаватися із вказанням кількості чисел або мати якусь ознаку закінчення.

Приклад 7.4. Ввести шість дійсних чисел та обчислити найбільше з них.

Алгоритм обчислення максимального числа послідовності:

1) ввести перше число `x`;

2) вважати, що воно є максимальним: `max = x;`

3) у циклі почергово вводити решту чисел. Кожне з уведених чисел порівнювати зі значенням `max` і, якщо число `x` буде більшим від `max`, запам'ятати його значення як `max` (знову `max = x;`):

```
if (x>max) max=x;
```

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int i;
double x, max;
cout << "Ввести 1 число: ";
cin >> x;
max = x;
for (i=2; i<=6; i++)
{
cout << "Ввести "<< i <<" число: ";
cin >> x;
if (x>max) max = x;
}
cout << "Найбільше число: " << max << endl;
system("pause");
return 0;
}
```

Результати виконання програми (для тестування задачі):

```
Ввести 1 число: 5.5
Ввести 2 число: 0.2
Ввести 3 число: -7
Ввести 4 число: 3.1
Ввести 5 число: 15.2
Ввести 6 число: -1.4
Найбільше число: 15.2
```

Приклад 7.5. Ввести 12 цілих чисел та обчислити добуток парних елементів з цієї числової послідовності.

Алгоритм розв'язання задачі буде полягати у такому. У циклі, який повторюватиметься 12 разів, виконувати такі дії:

– виводити запрошення для введення поточного числа;
– вводити поточне число;
– перевіряти введене число на парність (використано оператор `%` – остача від ділення цілих чисел) і, якщо це так, тоді число помножити на добуток. Окрім того, виконується перевірка чи поточне число не дорівнює нулю.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int i, x, dob=1;
// Спочатку добуток дорівнює 1.
for(i=1; i<=12; i++)
// У циклі 12 разів повторити дії:
{
cout << "Ввести " << i <<  " число: ";
// виводиться запрошення
cin >> x;
```

```
// введення поточного числа,
if((x%2==0) && (x!=0)) dob *= x;
// якщо число парне, перемножити його.
}
cout << "Добуток парних чисел: " << dob << endl;
system("pause");
return 0;
}
```

Результат виконання програмного коду:

```
Ввести 1 число: 3
Ввести 2 число: 1
Ввести 3 число: -7
Ввести 4 число: 6
Ввести 5 число: 3
Ввести 6 число: -2
Ввести 7 число: 4
Ввести 8 число: 7
Ввести 9 число: 11
Ввести 10 число: -4
Ввести 11 число: -5
Ввести 12 число: 3
Добуток парних чисел: 192
```

Поданий програмний код є цілком правильним, проте має один вагомий недолік. Якщо парних чисел немає, тоді результатом обчислення добутку буде виведено 1. Замість цього бажано вивести повідомлення, що парних чисел не було введено. Для цього потрібно організувати обчислення кількості введених парних чисел. Якщо після циклу ця кількість становитиме 0, тоді слід вивести відповідне повідомлення. Для обчислення кількості необхідно оголосити окрему змінну цілого типу `k`, якій спочатку присвоюється значення 0 (парних чисел ще нема). Якщо введене число є парним, кількість збільшиться на 1.

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int i, x, dob=1, k=0;
for(i=1; i<=12; i++)
{
cout << "Ввести " << i << " число: ";
cin >> x;
if((x%2==0) && (x!=0)) // Якщо число парне,
  { dob *= x; // накопичуватиметься добуток
k++;           // і кількість таких чисел.
  }
}
if(k > 0)
// Якщо кількість парних чисел є більшою за 0,
// виведеться повідомлення "Добуток парних
// чисел: "
cout << "\nДобуток парних чисел: " << dob <<
endl;
else
// інакше - виведеться повідомлення, що
// "Парних чисел немає."
cout << "\nПарних чисел немає." << endl;
system("pause");
return 0;
}
```

Результати виконання програмного коду:

```
Ввести 1 число: 3
Ввести 2 число: 1
```

```
Ввести 3 число: -7
Ввести 4 число: 5
Ввести 5 число: 3
Ввести 6 число: -7
Ввести 7 число: 3
Ввести 8 число: 7
Ввести 9 число: 11
Ввести 10 число: -17
Ввести 11 число: -5
Ввести 12 число: 3
Парних чисел немає.
```

Хоча в циклах `for` зазвичай використовується лише один лічильник, іноді можуть виникати ситуації, коли потрібно працювати одразу з кількома керуючими змінними. Для цього використовується оператор "Кома".

Наприклад, проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
int aaa, bbb;
for (aaa = 0, bbb = 9; aaa < 10; ++aaa, --bbb)
cout << aaa << " " << bbb << endl;
system("pause");
return 0;
}
```

Результат виконання програмного коду:

```
0 9
1 8
2 7
3 6
4 5
```

```
5 4
6 3
7 2
8 1
9 0
```

У заголовку циклу присвоюються початкові значення двом попередньо оголошеним змінним: `aaa = 0` і `bbb = 9` (з огляду на цю обставину, тип змінних `aaa` і `bbb` у заголовку циклу `for` відсутній, тип змінних оголошений перед початком циклу). З кожною ітерацією змінна `aaa` збільшується на одиницю, а `bbb` зменшується на одиницю.

7.3. Вкладені цикли `for`

Цикли можуть бути вкладеними один в одного. При використанні вкладених циклів треба проєктувати програмний код так, щоб внутрішній цикл повністю вкладався у тіло зовнішнього циклу, тобто цикли не повинні перетинатися (передавання керування перебігом виконання програмного коду з внутрішного циклу у зовнішній *заборонено*).

Своєю чергою, внутрішній цикл може містити власні вкладені цикли. Імена параметрів зовнішнього та внутрішнього циклів мають бути різними. Допускаються такі конструкції вкладених циклів:

```
for(int k=1; k<=10; k++)
{ . . .
for(int i=1; i<=10; i++)
  { . . .
for(int m=1; m<=10; m++)
     { ...
     }
  }
}
```

Приклад 7.6. Обчислити суму ряду

$$S = \sum_{i=0}^{7} \frac{2x^{2i-1}}{3(2i-1)!}$$

де $i = 1, 2, \ldots, 7$.

Обчислюючи суму s треба обчислити сім доданків, для обчислення кожного з яких потрібно сформувати вкладений цикл обчислення факторіалів (2i - 1)!. У поданому програмному коді кожний доданок обчислюється з використанням проміжної змінної `u`.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
double s=0, u, x;
int i, k, fakt;
cout << "Введіть значення x = ";
cin >> x;
for (i=1; i<=7; i++)
{
fakt = 1;
for (k=1; k<=2*i-1; k++) fakt *= k;
u = 2*pow(x,2*i-1)/(3*fakt);
cout << "\nДоданок "<<i<<"=" << u << endl;
s += u;
}
cout << "\nСума = " << s;
system("pause");
return 0;
}
```

Приклад 7.7. Обчислити

$$S = \sum_{i=0}^{m} \frac{i}{i-2} \prod_{k=1}^{i+1} \frac{k+3}{k}$$

значення m ввести з клавіатури.

З обчислень вилучити доданки і множники, які дорівнюють нулю в чисельнику або знаменнику.

У цьому прикладі програмного коду подані цикли є вкладеними один в одного, оскільки параметр внутрішнього циклу `k` залежить від параметра зовнішнього циклу `i` (`k` змінюється від `1` до `i+1`).

Добуток $\prod_{k=1}^{i+1}\frac{k+3}{k}$ є співмножником доданка і обчислюється у внутрішньому циклі змінною `dob`. Оскільки внутрішній цикл складається лише з однієї настанови, тому операторні дужки "`{ }`" не є обов'язковими.

Перед зовнішнім циклом для обчислення суми необхідно змінній `S`, в якій будуть накопичуватись доданки, присвоїти початкове значення 0 (нуль), а перед зовнішнім циклом для обчислення добутку змінній `dob` необхідно присвоїти значення `1` (одиниця).

Оскільки під час обчислення добутку беруть участь лише цілі числа, тому, щоб при діленні не втратити дробову частину, потрібно перетворити чисельник до дійсного типу. Для цього можна дописати крапку до числа 3 : `(k+3.0)/k`.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
      setlocale(LC_ALL, "Ukrainian");
int i, k, m;
cout << "Введіть значення m = ";
cin >> m;
double S = 0, dob;
for (i = 1; i <= m; i++)
if ( i != 2)
{ dob = 1;
for(k = 1; k <= i+1; k++)
```

```
dob*=(k+3.)/k;
S += i/(i-2.) * dob;
}
cout << "\nСума = " < S;
system("pause");
return 0;
}
```

7.4. Настанова циклу з передумовою

Для зручності проєктування програмних кодів, а не за необхідності, у С++ є три різні настанови циклу – `while`, `do-while` та `for`. З настановою `for` Ви уже мали нагоду ознайомитися.

Настанови з передумовою та з постумовою (післяумовою) використовуються для програмного реалізування циклічних обчислювальних процесів і є альтернативними до настанови `for`.

Циклічні, тобто повторювані, обчислення можна реалізувати за допомогою настанови циклу з передумовою (`while`-настанови). Вона має формат:

```
while (умова) <настанова>;
```

Слово `while` є зарезервованим, дужки обов'язкові, `while (умова)` – це заголовок циклу, а `<настанова>` – тіло (область дії). У цьому записі під елементом `<настанова>` розуміють або одну настанову, або блок настанов.

Механізм реалізування настанови циклу з передумовою полягає у наступному. Спочатку аналізується `умова` в заголовку циклу. Якщо `умова`, у результаті аналізу, набуває значення `true`, тоді виконується тіло циклу та знову аналізується `умова`. Якщо вона набуває значення `true`, тоді виконання тіла циклу повторюється. Виконання настанови циклу закінчується, коли, у результаті аналізу, `умова` набуває значення `false`. Отже, на останній ітерації циклу тільки аналізується `умова`, а тіло циклу не виконується.

Якщо під час першого аналізу `умова` набуває значення `false`. тоді настанови, які утворюють тіло циклу, не виконуються

жодного разу. Аналіз `умови` продовження виконання циклу і виконання після неї тіла циклу називають ітерацією циклу.

`Умову` в настанові циклу називають умовою продовження, оскільки, якщо вона істинна, тоді виконання настанов циклу продовжується. Цикл починається з обчислення `умови`, тому її називають передумовою.

Змінні, які змінюють значення у тілі циклу і так впливають на результат аналізу умови продовження циклу, називають параметрами циклу, або керуючими змінними. Цілочисельні параметри циклу, які змінюють значення з постійним кроком на кожній ітерації називають лічильником циклу.

Настанову циклу з передумовою застосовують зазвичай тоді, коли кількість повторень циклу наперед невідома, або немає явно вираженого кроку змінювання параметра циклу, а для багаторазових повторювань тіла циклу відомим є вираз `умови`, за істинності якої цикл продовжує виконання.

Ще раз наголосимо, що в тілі циклу варто передбачати змінювання параметрів, які формують результат аналізу `умови`, інакше умову виходу з циклу ніколи не буде виконано і відбуватиметься "зациклювання".

Структуру циклу з передумовою подано на рисунку 7.2.

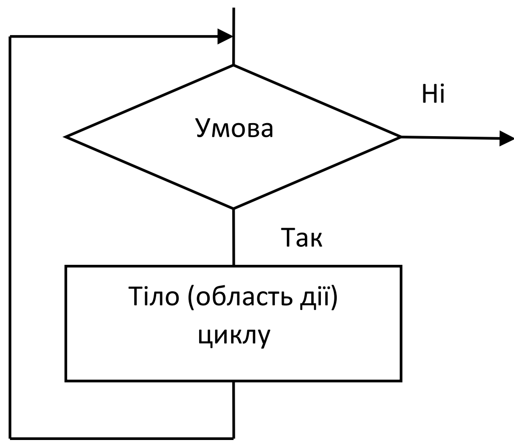


Рис. 7.2. Структуру циклу з передумовою

Розглянемо кілька фрагментів програмного коду.

Приклад 7.8.

```
d = 1; x = 4;
while (x <= 8)
{
d *= x; x++;
}
```

Значення параметру циклу змінюється на кожній ітерації циклу, тому цикл виконається 5 разів.

Приклад 7.9.

```
d = 1; x = 4;
while (x >= 8)
{ d *= x; }
```

Не задано зміну значення параметру циклу `x` і умову сформовано так, що вона набуває значення `false` за першого аналізу її значення, тому цикл не виконається жодного разу.

Приклад 7.10.

```
d = 1; x = 4;
while (x <= 8)
{ d *= x; }
```

Виникає ситуація "нескінченного циклу", оскільки параметр циклу `x` не змінює своє значення і `умова` у `while` не змінює свого значення.

Приклад 7.11.

Вивести усі числа від 0 до 9.

Проєкт програмного коду розв’язання задачі:

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

```
int main()
{
setlocale(LC_ALL, "Ukrainian");
int vne = 0;
while (vne < 10)
{
cout << vne << " ";
++ vne;
}
cout << "\nЗавершено!";
system("pause");
return 0;
}
```

Результат виконання програмного коду:

```
0 1 2 3 4 5 6 7 8 9
Завершено!
```

Розглянемо детально програму. По-перше, ініціалізується змінна `vne`:

```
int vne = 0;
```

Умова 0 < 10 набуває значення `true`, тому виконується тіло циклу. Першою настановою ми виводимо 0, а у другою – виконуємо оператор інкременту змінної `vne`. Потім керування повертається до початку циклу `while` для повторного аналізу умови. Умова 1 < 10 набуває значення `true`, тому тіло циклу виконується знову. Тіло циклу буде повторно виконуватися доти, доки змінна `vne` не дорівнюватиме 10, тільки тоді, коли результат аналізу умови 10 < 10 буде `false`, цикл завершиться.

Наприклад, тіло циклу `while` може взагалі не виконуватися, що підтверджує поданий проєкт програмного коду:

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

```
int main()
{
setlocale(LC_ALL, "Ukrainian");
int vne = 15;
while (vne < 10)
{
cout << vne << " ";
++ vne;
}
cout << "\nЗавершено!";
system("pause");
return 0;
}
```

Умова 15 < 10 відразу набуває значення `false` і тіло циклу пропускається. Єдине, що виведе ця програма:

```
Завершено!
```

Вкладені цикли `while`.

Одні цикли `while` можуть бути вкладені всередині інших циклів `while`. У прикладі 7.12 внутрішній та зовнішній цикли мають власні лічильники. Однак зауважте, що умова внутрішнього циклу використовує лічильник зовнішнього циклу!

Приклад 7.12. Проєкт програмного коду:

```
#include <iostream>
#include <iostream>
using namespace std;
int main()
{
int outer = 1;
while (outer <= 5)
{
int inner = 1;
```

```
while (inner <= outer)
cout << inner++ << " ";
cout << "\n";
++outer;
}
system("pause");
return 0;
}
```

Результат виконання програмного коду:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

7.5. Настанова циклу з постумовою do-while

Одна цікава річ у циклі `while` полягає в тому, що якщо результат аналізу умови циклу спочатку набуває значення `false`, тоді тіло циклу не виконуватиметься взагалі.

Проте іноді трапляються випадки, коли потрібно, щоб цикл виконався хоча б один раз. Для вирішення цієї проблеми C++ надає цикл `do-while`.

Настанова циклу з постумовою, (`do-while` настанова), має формат:

```
do {
<настанова>
}
while (умова);
```

Слово `do` (виконати) є ключовим. Хоча фігурні дужки є необов’язковими, якщо елемент `настанова` складається тільки з однієї настанови, проте їх часто використовують для поліпшення

візуального сприйняття конструкції `do-while`, не допускаючи плутанини з циклом `while`.

Настанова циклу з постумовою виконується так. Спочатку виконується `настанова`, яка формує тіло циклу, а потім аналізується `умова`. Якщо `умова`, у результаті аналізу, набуває значення `false`, тоді цикл завершується, інакше повторюється тіло циклу і знову аналізується `умова`. Настанова `do-while` завжди *починається виконанням тіла циклу* і закінчується аналізом `умови`.

Відмінність циклу з передумовою від циклу з постумовою полягає у виконанні першої ітерації: тіло циклу з постумовою завжди виконується принаймні одноразово незалежно від результату аналізу `умови`.

Настанови циклу з постумовою є в багатьох мовах програмування, але в деяких постумова розуміється як умова завершення (а не продовження) циклу. Тому, під час перекладу таких настанов іншими мовами *можуть виникати непорозуміння*.

Кожен цикл із постумовою можна замінити циклом з передумовою, який на відміну від настанови `do-while` *в усіх мовах програмування трактується однаково*.

Структуру циклу з постумовою подано на рисунку 7.3.

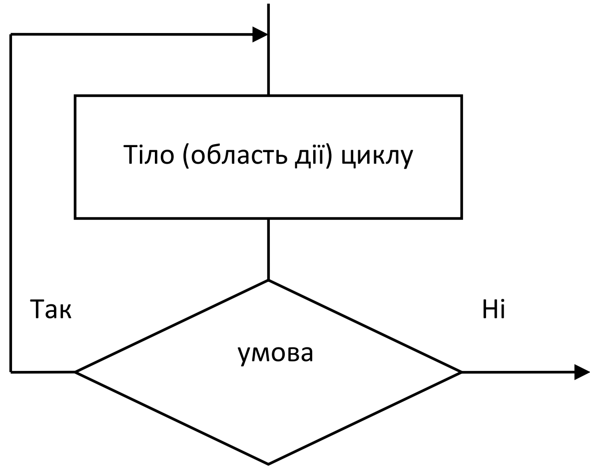


Рис. 7.3. Структура циклу з постумовою

Подамо відмінності у функціонуванні різних настанов циклу на прикладі обчислення суми всіх непарних чисел у діапазоні значень від 10 до 100:

1) з використанням настанови `for`

```
int i, s=0;
for (i=11; i<100; i += 2) s += i;
```

2) з використанням настанови `while`

```
int s=0, i=11;
while (i<100) { s += i; i += 2; }
```

3) з використанням настанови `do-while`

```
int s=0, i=11;
do { s += i; i += 2;} while (i<100);
```

Приклад 7.13. Використання циклу `do-while` для обчислення суми цілих чисел у заданому інтервалі [a; b], які змінюються з кроком 1.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int a;
cout << "Введіть значення початку інтервалу a: 
";
// початкове значення з інтервалу
cin >> a;
int b;
cout << "Введіть значення кінця інтервалу b: ";
// кінцеве значення з інтервалу
cin >> b;
```

```
int sum = 0;
int count = a;
do
{
sum += count;
count++;
}
while (count <= b);
// завершення циклу do while
cout << "\nsum = " << sum << endl;
system("pause");
return 0;
}
```

Приклад 7.14. Вводити цілі числа, доки не буде введене трицифрове число і обчислити кількість введених чисел та суму чисел, які належать до інтервалу [– 4, 11] з використанням настанови `do-while`;

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int x, k = 0, s = 0;
do {
cout << "Ввести поточне число: ";
cin >> x;
k++;
if (x >= -4 && x < 11) s += x;
}
while (abs(x) < 100 || abs(x) > 999);
cout << "Введено чисел: " << k << endl;
cout << "Сума чисел з проміжку [-4, 11]= " << s
<< endl;
system("pause");
return 0;
}
```

з використанням настанови `while`;

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int x = 0, k = 0, s = 0;
while (abs(x)<100 || abs(x)>999)
// Доки x не є трицифрове число
{
cout << "Ввести число: ";
cin >> x;
k++;
// збільшується кількість введених чисел,
if (x>=-4 && x<11)
// якщо x належить проміжку [-4, 11),
s += x;
// додається його значення до суми
}
cout << "\nВведено чисел: " << k << endl;
cout << "Сума чисел з проміжку [-4, 11)=" << s
<< endl;
system("pause");
return 0;
}
```

Переривання та продовження циклу.

Уперше настанову `break` було подано у настанові `swich`, де за її допомогою закінчувалося виконання настанови-перемикача.

Виконання настанови `break` всередині циклу довільного різновиду перериває і завершує цикл (далі виконуються настанови, які записані відразу за цим циклом).

Якщо `break` записано у настанові циклу, яка є вкладеною в іншу настанову циклу, тоді виконання `break` завершує вкладений цикл, а зовнішній цикл продовжується.

Настанова `continue` всередині циклу задає перехід на кінець тіла циклу. У настановах циклу з перед- і постумовою після `continue` аналізується `умова` продовження циклу.

РОЗ-
ДІЛ
8

СТРУКТУРОВАНІ ТИПИ ДАНИХ. МАСИВИ

Структуровані типи даних визначають впорядковану сукупність (послідовність) скалярних змінних і характеризуються типом своїх компонент. У С++ допускаються такі структуровані типи даних: рядки, масиви, множини, записи, файли а також процедурні типи та об'єкти.

Масив – це структурований тип даних, який складається з логічно впорядкованої, фіксованої кількості елементів однакового типу об'єднаних спільним іменем. Масиви використовуються для групування взаємопов'язаних змінних.

Елементами масиву можуть бути дані довільного типу, зокрема й структуровані. Тип елементів масиву називається базовим. Кількість елементів масиву фіксується під час оголошення і у процесі виконання програми не змінюється. Доступ до кожного окремого елемента масиву здійснюється шляхом індексування елементів (за значенням індексу елемента масиву). Індекс елемента масиву за змістом тотожний поняттю індекса елемента вектора. Індекси є виразами довільного скалярного типу, окрім дійсного.

8.1. Одновимірні масиви

Загальна форма оголошення одновимірного масиву:

```
<тип> <ім'я_масиву>[розмір];
```

де:
`тип` – це тип елементів масиву (базовий тип);
`розмір` – кількість елементів у масиві;
`ім'я_масиву` – (ідентифікатор) ім’я масиву, за яким здійснюється доступ до елементів масиву.

Загальний розмір масиву в байтах дорівнює розміру базового типу помноженого на кількість елементів (розмір масиву).

Після оголошення масиву, значення елементів масиву можуть бути нульовим або невизначеними (випадковими).

Це означає, що, наприклад, для масиву, який складається з 5 значень типу `int`, не потрібно створювати 5 окремих змінних, де кожній буде створено власний ідентифікатор. Натомість, ми можемо зберігати 5 різних значень одного типу, під одним унікальним ідентифікатором. Наприклад, структура масиву `array` може бути подана так (рис. 8.1):

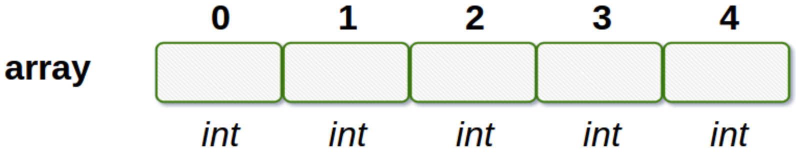


Рис. 8.1. Структура одновимірного масиву `array`

Оголошення масиву з 5 цілих чисел (тип `int`) з іменем `array`

```
int array[5];
```

Індекс у прямокутних дужках після імені масиву при його оголошенні вказує на розмір масиву, кількість елементів у масиві. Доступ до окремого елемента масиву здійснюється за ідентифікатором масиву та його індексом, який вказує на позицію (порядковий номер) елемента усередині масиву.

У С++ перший елемент масиву має нульовий індекс. Оскільки масив `array` містить 5 елементів, тому індекси його елементів змінюються від 0 до 4.

Ініціалізувати масив, тобто надати елементам масиву початкові значення можна різними способами: у процесі його оголошення або у процесі виконання програмного коду.

Під час оголошення одновимірного масиву підтримується два види ініціалізування числових значень елементів:

- ініціалізування з визначенням розміру;
- "безрозмірне" ініціалізування (без визначення розміру).

Загальний вигляд ініціалізування з визначенням розміру масиву:

```
<тип> <ім'я_масиву> [розмір] = {перелік_значень};
```

де:
`тип` – тип елементів масиву;
`розмір` – кількість елементів масиву вказаного типу;
`перелік_значень` – перелік значень ініціалізування елементів масиву.

Числові значення елементів масиву відділяються один від одного символом "," (кома).

Наприклад (рис. 8.2):

```
int array[5] = {1, 2, 4, 8, 16};
```

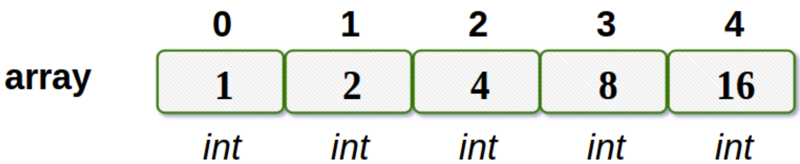


Рис. 8.2. Ініціалізування елементів масиву `array` *з визначенням розміру масиву*

Загальний вигляд "безрозмірного" ініціалізування:

```
<тип> <ім'я_масиву> [] = {перелік_значень};
```

У цьому випадку розмір масиву визначається кількістю елементів, числові значення яких оголошені в `переліку_ значень`.

Наприклад:

```
float Array1 [5];
// оголошено масив
float Array1 [] = {1, 4., 8, 5.5, 3};
// оголошено масив, автоматично визначено
// розмір, ініціалізовано числові начення
// елементів
float Array1 [5] = {2., 7, 9.6};
// оголошено масив, частково ініціалізовано
// числові значення елементів
float Array1 [5] = {2.6, 7, 9., 1.3, 6};
// оголошено масив, повністю ініціалізовано
// числові значення елементів
```

За замовчуванням, якщо в оголошеному масиві ініціалізується тільки значення декількох перших елементів, тоді його наступним елементам присвоюється значення 0 (нуль).

Так, у разі, коли

```
float Array2[10]= {2.2,34.56};
```

останнім восьмим елементам масиву присвоюється значення 0 (нуль). Тобто, якщо реальна кількість ініціалізованих значень є меншою від розміру масиву, тоді решті елементів масиву присвоюється значення 0 (нуль).

Для того, щоб усім елементам оголошеного масиву присвоїти значення 0 достатньо ініціалізувати його перший елемент значенням 0:

```
float Array2[0]={0};.
```

Доступ до окремих елементів масиву і операції з ними здійснюються так, ніби це звичайна змінна, яку можна модифікувати і зчитувати. Формат доступу до n-го елементу масиву виглядає так:

```
array[n];
```

Щоб отримати доступ до значення елемента масиву за індексом, достатньо вказати потрібний номер елемента в квадратних дужках. Наприклад, першим елементом масиву `array` розміром 5 є `array[0]`, а останнім – `array[4]`.

Якщо ми спробуємо доступитися до шостого елемента масиву `array`, оскільки елементів всього п'ять, ми вийдемо за межі масиву. У C++ це *не вважається* синтаксичною помилкою і компілятор нічого не повідомить, а така помилка виявиться лише у момент виконання програмного коду.

Наприклад: для зміни значення третього елемента масиву, потрібно записати:

```
int array2 [5] = { 1, 2, 0, 8, 16 };
array2[2] = 4;
// присвоюємо значення 4 для
// третього елемента масиву
int a = array[2];
// значення третього елементу масиву
// присвоюємо змінній а
```

Якщо кількість значень для ініціалізування елементів масиву є більшою від оголошеного розміру, тоді компілятор повідомить про помилку, якщо замалою – згідно з стандартом, неініціалізовані елементи заповняться нулями. Можливе ініціалізування окремих елементів масиву, наприклад:

```
int array[5] = {[2]=1}; //{0, 0, 1, 0, 0};
```

Для визначення розміру масиву можна вживати лише константи – явні або іменовані, такі які може обчислювати компілятор,

а також константні вирази. Не можна використовувати константи, які не можна обчислити під час компілювання, а також змінні. Наприклад:

```
const int n = 10;     // явна константа
int firstArr[n];      // OK
const int n1 = n+3;   // константний вираз
int secondArr[n1+2];  // OK: розмір може бути
                      // обчислений
                      // компілятором
```

Розглянемо поданий далі програмний код для ініціалізування значень елементів масиву `Array` цілого типу, який складається з десяти елементів квадратами чисел від 1 до 10.

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
      setlocale(LC_ALL, "Ukrainian");
int Array[10]; // Оголошення масиву Array
// Формуємо значення елементів масиву.
for(int t=0; t<10; ++t) Array[t] = (t+1)*(t+1);
// Візуалізування масиву
cout << "\nВивід сформованого масиву" << endl;
for(int t=0; t<10; ++t) cout << Array[t] << " ";
system("pause");
return 0;
}
```

У C++ всі елементи масиву займають суміжні елементи пам’яті. Інакше кажучи, елементи масиву в пам’яті розташовані послідовно один за одним. Комірка з найменшою адресою належить до першого елемента масиву, а з найбільшою – до останнього.

Приклад 8.1. Створити масив `Array` з 5 елементів цілого типу. Значення елементів ввести з клавіатури.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int SIZE;
cout << "\nВвести розмір масиву SIZE=";
cin >> SIZE;
int Array[SIZE];
for (int i = 0; i < SIZE; i++)
{
cout << "Введіть " << i + 1 << " елемент масиву\n";
cin >> Array[i];
}
for (int i = 0; i < SIZE; i++)
cout << "Array["<< i<< "] = " << Array[i] << endl;
system("pause");
return 0;
}
```

У поданому прикладі виведення елементів масиву виконується поелементно за допомогою настанови циклу.

З метою тестування програм часто використовують заповнення масивів випадковими (псевдовипадковими) числами. Випадкове число (випадкова величина, випадкове значення) – це число, значення якого не можна заздалегідь обчислити, тільки можна передбачити ймовірність потрапляння у певний діапазон. Послідовність випадкових чисел, отриманих з одного джерела, зазвичай мають спільні характеристики. У програмуванні найбільш поширеними є рівномірно розподілені випадкові числа – кожне значення однаково ймовірне у межах діапазону.

Псевдовипадкові числа, які зазвичай генеруються програмами, у своїй послідовності відтворюють характеристики випадкових чисел, але їх значення кожного разу повторюються.

У багатьох задачах (від моделювання природних або соціальних процесів до розкладання карт) потрібні послідовності чисел, які належать певній множині, але більше ніяк не пов'язані одне з одним. Такі числа називаються випадковими. Послідовності випадкових чисел часто імітують, використовуючи генератор псевдовипадкових чисел – підпрограму, яка за певним алгоритмом утворює послідовність чисел, що виглядає випадковою. Щоб кожного разу отримувати різні послідовності, алгоритм ініціалізують числом, яке зазвичай визначає всю послідовність.

Мовою C++ послідовності псевдовипадкових чисел можна отримувати за допомогою двох функцій, оголошених у файлі `cstdlib`. Функція з прототипом

```
void srand(unsigned int);
```

установлює початкове значення послідовності, використовуючи для цього аргумент у виклику (ним має бути невід'ємне ціле число).

Функція з прототипом

```
int rand(void);
```

обчислює й повертає наступне псевдовипадкове значення. Усі значення є цілими числами в діапазоні від 0 до 32767. Число 32767 позначається константою `RAND_MAX`, яку теж оголошено у файлі `cstdlib`.

Ім'я `rand` є скороченням слова *random* (випадковий), а `srand` – скороченням *seed random* (засіяти випадкове). Значення аргументу у виклику `srand` називається *зерном*, оскільки "зерно" є головним значенням англійського слова *seed*.

Розглянемо приклад програми, яка отримує від користувача зерно, створює тисячу псевдовипадкових дійсних чисел у діапазоні [0; 1] та обчислює їх середнє арифметичне (воно має бути доволі

близьким до 0,5). Дійсні числа з [0; 1] утворюються шляхом ділення цілих псевдовипадкових чисел на `RAND_MAX`.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
    setlocale(LC_ALL, "Ukrainian");
double sum;
int k, seed;
cout << "Введіть значення int seed"<<endl;
cin >> seed;
srand(seed);
for(k=0,sum=0; k<1000; ++k)
sum += rand()/double(RAND_MAX);
cout << "\nСереднє значення 1000 згенерованих
випадкових чисел = ";
cout << sum/1000 << endl;
system("pause");
return 0;
}
```

Приклад 8.2. Заповнити і вивести масив `randomArr` псевдовипадковими числами у діапазоні від 0 до 19 включно. Це приклад стандартного підходу до формування масиву та виведення його у консольному додатку.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
```

```
int n;
cout << "Введіть розмір масиву n=";
cin >> n;
int randomArr[n];
for (int i = 0; i < n; i++)
randomArr[i] = rand()%20;
for (int i = 0; i < n; i++)
cout << randomArr[i] << " ";
system("pause");
return 0;
}
```

Після кожного запуску програмного коду результати будуть однаковими.

Приклад 8.3. Створити і заповнити випадковими числами від 1 до 100 масив з 10 цілих елементів. Спочатку вивести на екран створений масив, а потім тільки парні елементи та їх індекси.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int SIZE;
cout << "\nВведіть розмір масиву SIZE=";
cin >> SIZE;
int Array[SIZE];
// Присвоюємо елементам масиву випадкові числа.
for (int i = 0; i < SIZE; i++)
Array[i] = 1 + rand ()%100;
// Виведення всього масиву:
cout << "Всі елементи масиву:" << endl;
```

```
for (int i = 0; i < SIZE; i++)
cout << Array[i]<<" ";
// Виведення парних елементів та їх індексів:
cout << "\nПарні елементи та їх індекси:" <<
endl;
for (int i = 0; i < SIZE; i++)
if (Array[i] % 2 == 0)
cout << " Array["<< i+1<< "] = " << Array[i] <<
endl;
system("pause");
return 0;
}
```

Проілюструємо фрагментом програмного коду зміну значень змінних під час обчислення максимального елемента масиву `A[10]` на кожній ітерації, які занесені у таблиці 8.1.

```
int A[10] = {3,7,2,4.9.11,4,3,6,3};
int i,Max;
for (i=0,Max=A[0]; i<10; i++)
if (A[i]>Max) Max=A[i];
```

Таблиця 8.1

Зміна значень змінних при обчисленні максимального елемента масиву A[10] на кожній ітерації

`i`	`A[i]`	`Max` до **`if`**	`Max` після **`if`**	Значення логічного виразу
0	3	3	3	**`false`**
1	7	3	7	**`true`**
2	2	7	7	**`false`**
3	4	7	7	**`false`**
4	9	7	9	**`true`**
5	11	9	11	**`true`**
6	4	11	11	**`false`**
7	3	11	11	**`false`**
8	6	11	11	**`false`**
9	3	11	11	**`false`**

Приклад 8.4. Створити і заповнити випадковими числами масив `Array` з 10 цілих елементів. Обчислити мінімальне та максимальне значення серед елементів масиву.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int min, max, t;
int Array[t];
cout << "\nВведіть розмір масиву t=";
cin >> t;
// Присвоюємо елементам масиву
// випадкові числа.
for(int i=0; i<t; i++) Array[i] = rand();
// Відтворюємо утворений масив;
for(int i=0; i<t;i++) cout<<Array[i]<< " ";
cout << endl;
//Обчислюємо мінімальне значення.
min = Array[0];
for(int i=1; i<t; i++)
if(min > Array[i]) min = Array[i];
cout << "\nМінімальне значення: " << min <<
endl;
// Обчислюємо максимальне значення.
max = Array[0];
for(int i=1; i<t; i++)
if(max < Array[i]) max = Array[i];
cout << "\nМаксимальне значення: " << max <<
endl;
system("pause");
return 0;
}
```

8.2. Сортування елементів масиву

Однією з найпоширеніших задач, які призначені для впорядкування елементів масивів, є сортування. Існує багато різних алгоритмів сортування. Часто застосовується, наприклад, сортування перемішуванням і сортування методом Шелла. Відомий також алгоритм *Quicksort* (швидке сортування з розбиттям початкового набору даних на дві половини так, що довільний елемент першої половини впорядкований щодо довільного елемента другої половини). Проте найпростішим вважають алгоритм сортування методом бульбашок. Незважаючи на те, що бульбашкове сортування не відрізняється високою ефективністю (і справді, його продуктивність неприйнятна для сортування великих масивів), його цілком успішно можна застосовувати для сортування масивів малого розміру.

Алгоритм сортування методом бульбашок отримав свою назву від способу, використовуваного для впорядковування елементів масиву. Тут виконуються оператори логічних відношень, які повторюються, і, у разі потреби, міняються місцями суміжні елементи. Водночас елементи з меншими значеннями поступово переміщаються до одного кінця масиву, а елементи з більшими значеннями – до іншого. Цей процес нагадує поведінку бульбашок повітря в резервуарі з водою.

Бульбашкове сортування здійснюється шляхом декількох проходів масивом, під час яких, за потреби, здійснюється переміщення елементів, які опинилися "не на своєму місці". Кількість проходів, які гарантують отримання відсортованих елементів масиву, дорівнює кількості елементів у масиві, зменшеному на одиницю.

Приклад 8.5. У поданому прикладі реалізовано алгоритм сортування елементів масиву (цілого типу), який містить випадкові числа. Проєкт програмного коду заслуговує на детальний аналіз.

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
```

```
#include <cstdlib>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int Array[10];
int a, b, t, size;
cout<<"\nВвести розмір масиву size=";
cin>>size;
// Присвоюємо елементам масиву випадкові
// значення
for(t=0; t<size; t++) Array[t] = rand()%20;
// Відтворюємо початковий масив
cout << "Початковий масив: ";
for(t=0; t<size; t++)
cout << Array[t] << " ";
cout << endl;
// Реалізування методу бульбашкового сортування
for(a=1; a<size; a++)
for(b=size-1; b>=a; b--) {
if(Array[b-1]> Array[b]) {
// Елементи неврегульовані
// Міняємо елементи місцями.
t = Array[b-1];
Array[b-1] = Array[b];
Array[b] = t;
}
}
// Кінець бульбашкового сортування.
// Відтворюємо Результат сортування масиву.
cout << "Результат сортування масиву: ";
for(t=0; t<size; t++)
cout << Array[t] << " ";
cout << endl;
system("pause");
return 0;
}
```

Приклад 8.6. Сформувати масив a цілого типу, обчислити максимальний елемент та його номер у масиві. Утворити новий масив b за таким правилом: якщо елемент масиву A знаходиться ліворуч від максимального і є парним числом, тоді збільшити його удвічі, якщо праворуч від максимального, тоді залишити без змін.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int n;
cout << "\nВведіть розмір масиву A n=";
cin >> n;
int a[n], b[n];
int max, nmax;
for (int i = 0; i < n; i++)
{
cout << "Введіть " << i+1 << " елемент масиву A:
";
cin >> a[i];
b[i]=a[i];
}
max = a[0];
nmax = 0;
for (int i = 1; i < n; i++)
if (a[i] > max)
{ max = a[i]; nmax = i; }
cout<<"\nМаксимальний   елемент   "<<max<<"   номер
max елементу "<<nmax+1;
for (int i = 0; i < n; i++)
if (a[i] % 2 == 0 && i < nmax)
b[i] = 2 * a[i];
cout << "\nМасив B"<<endl;
for (int i = 0; i < n; i++)
```

```
cout << b[i] << " ";
cout << endl;
system("pause");
return 0;
}
```

Приклад 8.7. Обчислити всі позиції входження числа k в масиві M з 10 цілих чисел.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int t;
int M[t];
cout << "\nВвести розмір масиву t=";
cin >> t;
int index[t]; // масив позицій входження числа k
int k;  // задане значення k
int n;  // позиції входження числа k
// введення числа k
cout << "Ввести задане число k" << endl;
cin >> k;
// введення елементів масиву M
for(int i=0; i<t; i++)
{cout << "Ввести M[" << i+1 << "] елемент" <<
endl;
cin >> M[i];
}
// відтворюємо масив M;
cout << "\nВвести масив M" << endl;
for(int i=0; i<t; i++)
cout << M[i] << " ";
```

```
// визначення всіх позицій заданого
// числа k в масиві M
// пошук числа k в масиві M і формування
// масиву index
n = 0;
for (int i=0; i<t; i++)
if (k==M[i])
{
// число, яке дорівнює k знайдено
n++;
index[n-1] = i;
}
cout << "\nКількість входжень заданого числа ="
<< n << endl;
// вивід результату
cout << "\nПозиції входження числа k у масиві "
<< endl;
for (int i=0; i<n; i++)
cout << index[i] << endl;
system("pause");
return 0;
}
```

8.3. Організація контролю меж масивів

У C++ не здійснюється жодної перевірки порушення контролю меж масивів, тобто ніщо не може перешкодити програмісту звернутися до значень елементів масиву, які розміщені за межами оголошеного розміру. Якщо це відбувається у процесі виконання настанови присвоєння, тоді можуть бути змінені значення в елементах пам'яті, виділених для зберігання значень інших змінних або навіть Вашій програмі. Інакше кажучи, звернення до значень елементів масиву (розміром у N елементів) за межею N-го елемента може призвести до руйнування програми за відсутності жодних зауважень з боку компілятора і без оголошення повідомлень про помилки під час роботи програми.

Це означає, що вся відповідальність за дотримання "кордонів" масивів покладається тільки на програмістів, які повинні гарантувати коректну роботу з масивами. Тобто, програміст зобов’язаний використовувати масиви достатньо великого розміру, щоб у них можна було без ускладнень поміщати дані. Однак найкраще у програмі передбачити перевірку перетину "кордонів" масивів.

Ця обставина пояснюється тим, що мова програмування C++ орієнтована на використання фаховими програмістами і з пункту бачення розробника нічого дивного у цьому нема.

8.4. Двовимірні масиви

Багатовимірні масиви можна оголосити як "масиви масивів". Наприклад, двомірний масив варто розглядати як таблицю елементів, які мають однаковий тип даних.

Формат оголошення двовимірного масиву має такий вигляд:

```
<тип> <ім’я> [ <розмір1>] [ <розмір2>];
```

Приклад структури оголошеного масиву `Myarray`, який має 3 рядки і 5 стовпців подано на рисунку 8.3:

```
int Myarray [3][5];
```

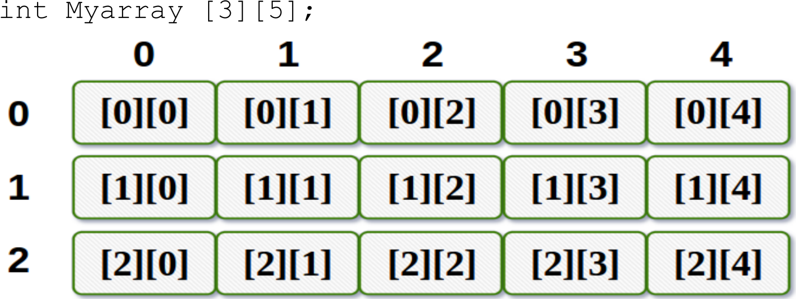


Рис. 8.3. Структура двовимірного масиву `Myarray`

Двовимірні масиви є впорядкованим записом декількох одновимірних масивів. Розташування кожного елемента визначається за допомогою двох індексів – номера рядка і номера стовпця, тобто порядкового номера у рядку і у стовпці. Нумерування індексів двовимірних масивів починається з нуля.

Кількість елементів масиву дорівнює добутку кількості елементів за кожним індексом.

Двовимірний масив задається або переліком елементів у тому порядку, в якому вони розташовані у пам'яті, або подається як масив масивів, кожний з яких записується в своїх фігурних дужках ″{ }″.

Під час оголошення й одночасного ініціалізування значень елементів двовимірних масивів можна ігнорувати кількість індексів тільки першого розміру. Якщо ініціалізування не здійснюється під час оголошення масиву, тоді кількість індексів треба вказувати явно.

Для доступу до елементів двовимірного масиву потрібно вказати ім'я та значення двох індексів. Якщо розглядати масив як таблицю, тоді перший індекс визначає номер рядка, а другий – номер стовпця таблиці.

Оголошення двовимірного масиву `Matr` цілих чисел розміром 3×4.

```
// двовимірний масив розміром 3×4
int Matr[3][4];
//ініціалізування значень елементів масиву
Matr[0][0] = 23;
Matr[2][3] = 41;
Matr[1][2] = -8;
```

Зауважте на подане у фрагменті програмного коду оголошення. На відміну від багатьох інших мов програмування, у яких під час оголошення масиву значення розмірів відокремлюються комами, у мові програмування С++ кожний розмір записаний у *власну пару квадратних дужок*.

Щоб присвоїти усім елементам двовимірного масиву `Matr` значення 0 (нуль), потрібно написати фрагмент такого програмного коду:

```
int  Matr[3][4];  //  двовимірний  масив  розміром
                  // 3 × 4
int i,j;
for (i=0; i<3; i++)
    for (j=0; j<4; j++)
        Matr[i][j] = 0;
```

Приклад ініціалізування значень елементів двовимірного масиву M розміром 3×4 дійснийсного типу.

```
// ініціалізування масиву M дійсних чисел
// розміром 3×4
float M[3][4] =
{
   { 0.5, -2.8, -1.0, 23.45 },
   { -2.3, 0.4, 10.5, 0.8 },
   { 12.5, 10.4, 5.4, 3.56 }
};
```

Якщо у переліку вказати не всі елементи групи, тоді відсутні елементи будуть доповнюватись нульовими значеннями автоматично:

```
// ініціалізування масиву M дійсних чисел
// розміром 3 × 4
// відсутні елементи доповнюються нулями
float M[3][4] =
{
   { 0.5, -2.8 },
   { -2.3 },
   { 12.5, 10.4, 5.4, 3.56 }
};
```

Приклад ініціалізування значень масиву B цілих чисел без визначення розміру:

```
// Ініціалізування значень масиву B
// без визначення розміру
```

```
int B[][4] =
{
   {  2, -8,  3,   4 },
   { -3, 50,  42, -77 },
   { 11, 25, -30,  4 }
};
```

Приклад ініціалізування значень масиву B цілих чисел без визначення розміру з доповненням нулями:

```
// Ініціалізування значень масиву B
// без визначення розміру
// з доповненням нулями
int B[][4] =
{
    { 2, -8 },
    { -3 },
    { 11, 25, -30, 4 }
};
```

Як уже згадувалося, другий індекс масиву (вказує на номер стовпця) обов'язково має бути вказаний. В іншому разі компілятор повідомить про помилку.

Розглянемо кілька прикладів типових підходів до розв'язання задач впорядкування елементів двовимірних масивів. Для цього використовуються вкладені структури параметричної настанови циклу. Нагадаємо, що параметр завнішнього циклу не змінить свого значення, доки параметр внутрішнього циклу не змінить своїх значень від початкового до кінцевого з визначеним кроком.

У прикладах подано типові підходи до введення/виведення матриць рядками. Типові підходи до накопичення суми або добутків уже розглядалося у попередніх розділах.

Приклад 8.8. У матриці A(n, n), (n<=9) обчислити суму всіх додатних елементів, крім тих, які знаходяться на головній діагоналі.

Особливістю елементів матриці, які розміщені на головній діагоналі, є те, що значення індексів цих елементів однакові. Суть задачі зводиться до формування і аналізу умови у настанові умовного переходу

```
if (i!=j && C[i][j]>0) sd += C[i][j];
```

де аналізуються значення індексів поточного елемента і його числове значення. Накопичення суми відбувається тоді, коли значення індексів поточного елемента не дорівнюють один одному і значення цього елемента є більшим від нуля. Усе решта є типовими підходами у програмуванні, тобто ″технічними″.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int i, j, n;
cout << "Введіть порядок матриці n (<=9) = ";
cin >> n;
float C[n][n];
cout << "\n Введіть значення елементів матриці
C" << endl;
for(i=0; i<n; i++)
for(j=0; j<n; j++)
{
cout << "C[" << i << "][" << j << "]=";
cin >> C[i][j];
}
float sd=0;
for (i=0; i<n; i++)
for (j=0; j<n; j++)
if (i!=j && C[i][j]>0)
sd+=C[i][j];
```

```
cout << "\n сума = " << sd << endl;
system("pause");
return 0;
}
```

Приклад 8.9. Матриця A(m, n) (m<=9, n<=9) містить додатні та від'ємні елементи. Сформувати одновимірні масиви: B, який містить лише додатні елементи масиву A(m, n), масив C, який містить лише від'ємні елементи масиву A(m, n). Обчислити кількість елементів в даних масивах.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int i, j, m, n, k;
cout << "Введіть к-сть рядків матриці m (<=9) =
";
cin >> m;
cout<<"Введіть к-сть стовпців матриці n (<=9) =
";
cin >> n;
float A[m][n];
k=m*n; // Розмірність масивів B і C
float B[k], C[k];
// Формування матриці A
for(i=0; i<m; i++)
for(j=0; j<n; j++)
{
cout << "A[" << i << "][" << j << "]=";
cin >> A[i][j];
}
int kd=0, kv=0;
for (i=0; i<m; i++)
```

```
for (j=0; j<n; j++)
if (A[i][j]>0)
{
B[kd] = A[i][j];
kd++;
}
else if (A[i][j]<0)
{
C[kv] = A[i][j];
kv++;
}
cout << "кількість додатних =" << kd << endl;
for(i=0; i<kd; i++)
cout << "B[" << i << "]=" << B[i] << endl;
cout << "кількість від'ємних =" << kv << endl;
for(i=0; i<kv; i++)
cout << "C[" << i << "]=" << C[i] << endl;
system("pause");
return 0;
}
```

Приклад 8.10. Задано масив цілих чисел A[n , n]. Визначити мінімальний елемент 3-го стовпця та мінімальний елемент масиву.

Розв'язуючи цю задачу, потрібно пам'ятати, що у елементів третього стовпця перший індекс буде змінюватися від `0` до `n`, а другий індекс буде дорівнювати `2` (нагадуємо, нумерування індексів у С++ починається від нуля).

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
int n;
cout<<"Введіть порядок матриці n (<=9) = ";
```

```
cin >> n;
int A[n][n], min;
for (int i=0; i<n; i++)
{
for (int j=0; j<n; j++)
{
A[i][j] = rand() % 51;
cout << A [i][j] << "\t";
}
cout << endl;
}
min=A[0][2];
for (int i=1; i<n; i++)
if (A[i][2] < min) min = A[i][2];
cout  <<  "Мінімальний  елемент  3  стовпця  =  "  <<
min << endl;
min=A[0][0];
for (int i=0; i<n; i++)
for (int j=0; j<n; j++)
if (A[i][j] < min) min = A[i][j];
cout << "Мінімальний елемент матриці = " << min
<< endl;
system("pause");
return 0;
}
```

Приклад 8.11. Створити матрицю A(5, 5), елементи на діагоналі якої є одиниці, над головною діагоналлю – 10, а під головною діагоналлю – 100.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
int n;
cout << "Введіть порядок матриці n (<=9) = ";
```

```
cin >> n;
int A[n][n];
for (int i=0; i<n; i++)
A[i][i] = 1;
for (int i=0; i<n; i++)
for (int j=i+1; j<n; j++)
A[i][j] = 10;
for (int i=0; i<n; i++)
for (int j=0; j<i; j++)
A[i][j] = 100;
for (int i=0; i<n; i++)
{
for (int j=0; j<n; j++)
cout << A[i][j] << "\t";
cout << endl;
}
system("pause");
return 0;
}
```

Приклад 8.12. Задано матрицю `A` цілих чисел розмірністю 5 × 7. Розробити проєкт програмного коду для обчислення вектора `X`, елементами якого є суми від'ємних елементів відповідних стовпців матриці.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
int A[5][7], X[7]={};
for (int i=0; i<5; i++)
{ for (int j=0; j<7; j++)
{ A[i][j] = rand() % 20 - 5;
cout << A[i][j] << "\t";
}
```

```
cout << endl;
}
cout << endl;
for (int j=0; j<7; j++)
{ for (int i=0; i<5; i++)
if (A[i][j] < 0)
X[j] += A[i][j];
cout << X[j] << "\t";
}
cout << endl;
system("pause");
return 0;
}
```

Приклад 8.13. З елементів матриці C розмірністю 4 × 7 дійсних чисел обчислити вектор добутків ненульових елементів парних (2, 4 та 6) стовпців матриці.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int k=0;
float C[4][7], p[3];
cout << "Введіть матрицю C(4 x 7): " << endl;
for (int i=0; i<4; i++)
{ for (int j=0; j<7; j++)
{ C[i][j] = rand() % 20 - 10;
cout << C[i][j] << "\t";
}
cout << endl;
}
```

```
cout << endl;
cout << "Вектор добутків ненульових "<< endl;
cout << "елементів парних стовпців матриці: " <<
endl;
for (int j=1; j<7; j+=2)
{ p[k] = 1;
for (int i=0; i<4; i++)
if (C[i][j] != 0) p[k] *= C[i][j];
cout << p[k] << "\t";
k++;
}
cout << endl;
system("pause"); return 0;
}
```

Приклад 8.14. Ввести матрицю B розмірністю 4×4 цілих чисел і поміняти місцями значення елементів головної діагоналі зі значеннями мінімальних елементів відповідних рядків місцями.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
const int n = 4;
int B[n][n], k, min;
cout << "\nМатриця B(4 x 4):" << endl;
for (int i=0; i<n; i++)
{ for (int j=0; j<n; j++)
{ B[i][j] = rand() % 5;
cout << B[i][j] << "\t";
}
```

```
cout << endl;
}
for (int i=0; i<n; i++)
{ min = B[i][0]; k = 0;
for (int j=1; j<n; j++)
if (B[i][j] < min)
{ min = B[i][j]; k = j; }
B[i][k] = B[i][i];
B[i][i] = min;
}
cout << "Перетворена матриця B: " << endl;
for (int i=0; i<n; i++)
{ for (int j=0; j<n; j++)
cout << B[i][j] << "\t";
cout << endl;
}
system("pause");
return 0;
}
```

Приклад 8.15. З елементів матриці А розмірністю 7 × 7 цілих чисел обчислити вектор скалярних добутків елементів першого рядка на стовпці матриці.

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
const int n = 7;
int A[n][n], p[n]={};
cout << "Формування матриці A(7x7): " << endl;
for (int i=0; i<n; i++)
{ for (int j=0; j<n; j++)
```

```
{ A[i][j] = rand() % 15;
cout << A[i][j] << "\t";
}
cout << endl;
}
cout << "Вектор скалярних добутків: " << endl;
for (int j=0; j<n; j++)
{ for (int i=0; i<n; i++)
p[j] += A[0][i] * A[i][j];
cout << p[j] << "\t";
}
cout << endl;
system("pause");
return 0;
}
```

Приклад 8.16. Дано квадратну матрицю A 6-го порядку. Обчислити суму всіх елементів матриці, які розміщені у рядках з від'ємним елементом на головній діагоналі. Обчислити кількість таких рядків.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
// Оголошення змінних та ініціалізування їх
// значень
int n;
cout << "Введіть порядок матриці A: ";
cin >> n;
int count=0;
int sum=0;
int A[6][6];
```

```
cout << "Формування матриці A(6x6): " << endl;
for(int i=0; i<n; i++)
for(int j=0; j<n; j++)
A[i][j]=-10+rand()%41;
//Виконання обчислень
for(int i=0; i<n; i++)
if(A[i][i]<0)
{
count++;
for(int j=0; j<n; j++)
sum+=A[i][i];
}
//Вивід матриці
for(int i=0; i<n; i++)
{
for(int j=0; j<n; j++)
cout << A[i][j] << "\t";
cout << endl;
}
cout << "К-сть рядків з від'ємними елементами на
головній діагоналі: " << count << endl;
cout << "Сума елементів цих рядків =" << sum <<
endl;
system("pause");
return 0;
}
```

Приклад 8.17. Задано матриці A(n, n), B(n, n). Розробити проєкт програмного коду обчислення суми C = A + B.

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int n;
```

```
cout << "Введіть порядок матриць A, B, C n (< =
9) = ";
cin >> n;
int A[n][n], B[n][n], C[n][n];
cout << "\nМатриця A" << endl;
for (int i=0; i<n; i++)
{ for (int j=0; j<n; j++)
{ A[i][j] = rand() % 6;
cout << A[i][j] << "\t";
}
cout << endl;
}
cout << "*****************" << endl;
cout << "\nМатриця B" << endl;
for (int i = 0; i<n; i++)
{ for (int j = 0; j<n; j++)
{ B[i][j] = rand() % 5;
cout << B[i][j] << "\t";
}
cout << endl;
}
cout << "*****************" << endl;
cout << "\nМатриця C" << endl;
for (int i = 0; i<n; i++)
{
    for (int j = 0; j<n; j++)
{
C[i][j] = A[i][j] + B[i][j];
cout << C[i][j] << "\t";
}
cout << endl;
}
system("pause");
return 0;
}
```

Приклад 8.18. Задано матриці A(n, n), B(n, n). Розробити проєкт програмного коду обчислення добутку C = A * B.

У загальному випадку, якщо матриця A має розмір $n \times m$, а матриця P – $m \times t$, тоді їхнім добутком буде матриця Q розміром $n\times t$, елементи якої обчислюють за формулою $\sum_{k=1}^{m} A_{ik} * P_{kj}$, i = 1, ..., n, j = 1, ..., t. Отже, для обчислення матриці Q доведеться використати три вкладені цикли: два – для перебору Q_{ij}, третій – для накопичення суми.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
int main()
{
setlocale(LC_ALL, "Ukrainian");
int n;
cout << "Введіть порядок матриць A, B, C n (<=9)
= ";
cin >> n;
int A[n][n], B[n][n], C[n][n] = {};
cout << "\nМатриця A" << endl;
for (int i=0; i<n; i++)
{ for (int j=0; j<n; j++)
{ A[i][j] = rand() % 3;
cout <<A[i][j] << "\t";
}
cout << endl;
}
cout << "*****************" << endl;
cout << "\nМатриця B" << endl;
for (int i=0; i<n; i++)
{ for (int j=0; j<n; j++)
{ B[i][j] = rand() % 5;
cout < <B[i][j] << "\t";
}
cout << endl;
```

```
}
cout << "*****************" << endl;
cout << "\nМатриця C" << endl;
for (int i=0; i<n; i++)
{ for (int j=0; j<n; j++)
{ for (int k=0; k<n; k++)
C[i][j] += A[i][k] * B[k][j];
cout << C[i][j] << "\t";
}
cout << endl;
}
system("pause");
return 0;
}
```

Розділ 9

МОДУЛЬНА ОРГАНІЗАЦІЯ ПРОГРАМ. ОРГАНІЗАЦІЯ ФУНКЦІЙ У С++

Часто Ваші програми будуть переривати виконання одних функцій для виконання інших. Ви робите аналогічні речі у реальному житті постійно. Наприклад, читаєте книгу і згадали, що мали зателефонувати. Залишаєте закладку у своїй книзі, берете телефон та набираєте номер. Після того, як Ви завершили розмову, Ви повертаєтеся до читання: до тієї сторінки, де зупинилися.

Програми мовою С++ працюють подібним способом. Іноді, коли програма виконує код, може зіткнутися з викликом функції. Виклик функції – це вираз, який вказує процесору перервати виконання поточної функції та розпочати виконання іншої функції. Процесор "залишає закладку" в поточному місці виконання, а потім виконує функцію, яка викликається. Коли виконання функції, яка викликається, завершено, процесор повертається до закладки і відновлює виконання перерваної функції. Функція, в якій знаходиться виклик, називається викликаючою функцією *(caller)*.

Мова С++ дає можливість реалізувати концепцію структурного аналізу алгоритмів. Структурний аналіз полягає у попередньому опрацюванні складної задачі або громіздкого алгоритму та поділі його на окремі простіші частини. У С++ це реалізується за допомогою функцій. Окремі функції об'єднують у спільну програму. У завершеному вигляді така програма утворює модуль. Крім того, функції можна багаторазово використовувати та переносити в інші програми.

Розрізняють стандартні функції та функції користувача. Стандартні функції описані в бібліотеках. Щоб скористатися стандартною функцією, треба під'єднати відповідну бібліотеку. Якщо бібліотека під'єднана, тоді у програмному коді можна викликати функцію.

Функція у програмуванні – це один з видів підпрограми. Особливість, яка відрізняє її від іншого виду підпрограм – процедури, полягає в тому, що функція повертає значення, а її виклик може використатися в програмі як вираз.

Підпрограма – частина програми, яка реалізовує певний алгоритм і дає змогу звернення до неї з різних частин загальної (головної) програми. Підпрограма часто використовується для скорочення розмірів програм у тих завданнях, у процесі розв'язання яких необхідно виконати декілька разів однаковий алгоритм при різних значеннях параметрів. Настанови (команди), які реалізують відповідну підпрограму, записують один раз, а в необхідних місцях розміщують настанови передавання керування на цю підпрограму.

З погляду теорії систем, функція в програмуванні – окрема система (підсистема, підпрограма), на вхід якої надходять керуючі впливи у вигляді значень аргументів. На виході системи одержуємо результат виконання програми, що може бути як скалярною величиною, так і векторним значенням. Заходом виконання функції можуть виконуватися також деякі зміни в керованій системі, причому і зворотні, і незворотні.

Отже, коли програма стає завеликою за обсягом і складною для сприймання, є сенс поділити її за змістом на невеликі логічні частини, підпрограми, названі функціями, кожна з яких виконуватиме певне завдання. Унаслідок цього програма стане більш легкою і для розуміння при створюванні, і для процесу налагодження.

Крім того, створення функції позбавляє потреби створювання двох, а іноді й більшої кількості, майже однакових фрагментів програмного коду для вирішення схожих завдань за різних вхідних даних.

Розділювання програми на функції є базовим принципом структурного програмування.

9.1. Організація функцій у C++

Функція – це незалежна іменована частина програми, яка може багаторазово викликатися з інших частин програми, маніпулювати даними та повертати результати. Кожна функція має власне ім'я, за яким здійснюють її виклик. Розрізнюють два основні різновиди функцій:

– стандартні вмонтовані функції, які є складовою мови програмування C++, наприклад: `sin()`, `pow()` тощо;

– функції, створювані користувачем для власних потреб.

Створювана у програмі функція повинна бути *оголошеною* і *означеною*. Оголошення функції має бути виконаним у програмі перед її використанням. Означення можна реалізувати у довільному місці програми, за винятком тіла (середини) інших функцій.

Нагадуємо, що з-поміж функцій програми повинна бути одна з ім'ям `main` (головна функція), яка може знаходитися у довільному місці програми. Ця функція виконується завжди першою і закінчується останньою.

Оголошення функції (прототип, заголовок) задає: ім'я функції; тип значення, яке повертає функція (якщо воно є); імена та типи аргументів, які можуть передаватися як у функцію, так і з неї.

9.1.1. Прототип функції

Ім'я функції необхідно оголосити в тексті програми до того, як воно буде використовуватися. Проте записувати всю функцію вище від її викликів не обов'язково – достатньо записати лише її заголовок.

Заголовок функції зі знаком `";"` у кінці називається ***прототипом функції***. Прототип є настановою оголошення функції і повідомляє компілятору, що в програмі є така функція. Після оголошення функцію все ж необхідно *означити*, тобто описати задані нею дії.

Кожен елемент програми (змінна, функція тощо) повинен мати ім'я, яким він позначається. *Перш, ніж користуватися елементом, необхідно його оголосити*.

Оголошення імені елемента лише описує його, але не створює сам елемент. Проте, щоб використовувати елемент програми, необхідно спочатку створити його в пам'яті програми. Створення елемента і задається його означенням.

Функція, сформована заголовком і тілом, є означенням, оскільки дії, задані функцією у вигляді машинних команд, записуються в певну ділянку пам'яті. Прототип же лише повідомляє про функцію і не описує жодних дій, тому є оголошенням її імені.

Оголошення функції має формат:

```
[клас] <тип результату> <ім'я функції>
   (<перелік_формальних_аргументів
      з оголошенням типу>);
```

де:

`тип результату` – довільний базовий або раніше описаний тип значення (за винятком масиву і функції), який повертається функцією (необов'язковий параметр). За відсутності цього параметра тип результату за замовчуванням буде цілий (`int`). Він також може бути описаний ключовим словом (**`void`**), тоді функція не повертає ніякого значення;

`ім'я функції` – ідентифікатор функції, за яким завжди є пара круглих дужок ″( )″, де записуються формальні параметри (аргументи). Фактично ім'я функції – це особливий вид вказівника на функцію, його значенням є адреса початку входу у функцію;

`перелік_формальних_аргументів` – визначає кількість, тип і порядок оголошення переданих у функцію вхідних аргументів (у літературі дуже часто використовується термін – *параметр*), які відділяються комою (″ , ″). Якщо функція не має аргументів, (елемент `перелік_формальних_аргументів` відсутній), тоді круглі дужки залишаються порожніми.

Перелік формальних аргументів має такий вигляд:

```
([const] тип1 [аргумент1],
[const] тип2 [аргумент2],...)
```

Отже, перелік формальних аргументів є послідовністю пар (складаються з типу даних і імені) відокремлених між собою комами.

Аргументи функції локалізовані і є недоступними для інших функцій. Якщо аргументи мають однаковий тип їх все одно не можна об'єднувати, всі параметри записуються окремо.

Прототип функції може бути розміщеним як безпосередньо у програмі, так і в заголовному файлі. У разі, коли означення функції розміщено у програмі раніше за точку виклику, писати окремо оголошення й окремо реалізування цієї функції немає потреби, тоді оголошення функції можна уникнути.

Після прототипів зазвичай записують головну функцію, а за нею – оголошені та інші функції. Порядок розташування оголошених функцій може бути довільним. Із прототипом функції варто вказати:

– призначення функції, тобто що саме вона виконує та яким є зміст її параметрів;

– *передумови* – умови, які мають справджуватися *перед* її викликом, зокрема умови для її аргументів (фактичних параметрів);

– *післяумови* – умови, які справджуються *після* її виклику, зокрема, як функція змінює глобальні змінні програми;

– поведінку функції за некоректних фактичних аргументів.

Пам'ятайте: прототип функції пишеться не тільки для компілятора, але й для програміста, який має використовувати функцію. Отже, прототип потрібно писати так, щоб, не читаючи тіла функції, можна було зрозуміти, як нею користуватися. Запишемо прототип функції обчислення довжини відрізка з відомими координатами початку і кінця.

```
double dist(double a1, double b1,
double a2, double b2);
// distance from (a1,b1) to (a2,b2)
```

У прототипі, на відміну від означення функції, можна не вказувати імен аргументів. Зокрема, прототип функції `dist` із погляду синтаксису може мати вигляд

```
double dist(double, double, double, double);
```

Проте він не дає інформації про призначення аргументів функції. Тому це радше приклад того, як *не варто* писати прототип функції, адже вдало названі параметри й сама функція дають змогу уникнути зайвих пояснень щодо її призначення. Замість скороченого імені `dist` краще було б узяти `distance` (*відстань*), але воно є зарезервованим в бібліотеках мови С++, тому залишимо `dist`.

У прототипах імена аргументів ігноруються компілятором, вони необхідні тільки для покращення візуального сприйняття програми.

9.1.2. Означення (реалізування) функції

Крім оголошення, кожна функція повинна мати означення (реалізування). Означення функції – це задання способу виконання операцій.

Функція, крім заголовку, містить тіло функції (настанови, які виконує функція) і настанову повернення результату `return`:

```
<тип функції> <ім'я функції> (<перелік_параметрів>)
{
<тіло функції>
return <результат обчислень>;
}
```

У фігурних дужках записано тіло функції. Тіло функції становлять С++-настанови, які визначають конкретні дії функції. Функція завершується (і керування передається процедурі, яка її викликає), досягши закритої фігурної дужки. Якщо результат повертається функцією, тоді в тілі функції є необхідною настанова
`return <результат обчислень>;`,
де `результат обчислень` формує значення, яке є результатом і збігається з його типом.

Фігурні дужки обмежують тіло функції (послідовності настанов), які розв'язують поставлену задачу. Крім того, фігурні

дужки є межами області видимості всіх змінних, оголошених усередині тіла функції (такі змінні називаються *локальними*). Ця область закрита від зовнішнього впливу і доступ до неї з інших функцій неможливий. Вся інформація може надходити ззовні лише двома каналами: через фактичні параметри, або аргументи, які є значеннями формальних параметрів; через глобальні змінні, які є видимими у довільній точці програми.

Змінні, які використовуються під час виконання функції, можуть бути глобальні та локальні. Змінні, які описані (визначені) за межами функції, називають глобальними. *За допомогою глобальних параметрів можна передавати дані у функцію, не включаючи ці змінні до складу формальних аргументів*. У тілі функції їх можна змінювати і потім отримані значення передавати в інші функції.

Змінні, які описані у тілі функції, називаються локальними або автоматичними. Вони створюються тільки на час роботи функції, а після реалізування функції система видаляє локальні змінні та звільняє пам'ять. Тобто, між викликами функції вміст локальних змінних знищується, тому ініціювання локальних змінних треба робити щоразу під час виклику функції. За необхідності збереження цих значень, їх треба описати як статичні за допомогою службового слова `static`, наприклад:

```
static int x, y;
```

або

```
static float s = 3.25;
```

Статична змінна схожа на глобальну, але діє тільки у тій функції, в якій вона оголошена.

Нагадаємо, на початку програми можна не описувати всю функцію, а записати тільки прототип. Запис прототипу може містити тільки перелік типів формальних параметрів без імен, а наприкінці прототипу завжди ставиться символ " ; ", тоді як у означенні функції цього символу після заголовку немає.

На прикладі розглянемо означення функції `seredne`. Це можна подати так:

```
double seredne(int a, int b)
{
return (a+b)/2.0;
}
```

Подана функція обчислює середнє арифметичне двох цілих чисел a та b і повертає його значення у точку виклику за допомогою настанови `return`.

Подамо фрагмент програмного коду, в якому виклик функції `seredne()` може бути реалізованим в один зі способів:

```
int a=5, x=2, y=-3;
double z, s1, s2;
z = seredne(5, 4);     // z = 4.5
s1 = seredne(a, 11);   // s1 = 8
s2 = seredne(x, y);    // s2 = -0.5
```

У першому поданому виклику числа 5 та 4 є фактичними аргументами, які записуються у дужках на місцях формальних аргументів `a` та `b`:

```
double seredne(int a, int b)
                    ↑      ↑
      z = seredne (5,      4);
```

У другому виклику першим фактичним параметром є змінна `a`, а другим – число 11. У третьому – обидва фактичних параметри є змінними `x` та `y`.

Якщо функції виконують певні обчислення й дії, які не потребують повернення результатів, замість їх типу вказують тип `void` (тобто порожній, без типу). У таких функціях настанова `return` може бути відсутньою або записуватись без значення, яке повертається.

9.2. Основні правила організації функцій

1. Кількість фактичних аргументів функції має збігатися з кількістю її формальних аргументів.

2. Типи змінних, які є фактичними аргументами, мають збігатися з типами відповідних формальних аргументів, або мати можливість бути перетвореними до типів формальних аргументів.

3. Імена змінних, які є фактичними аргументами, можуть не збігатися з іменами формальних аргументів.

4. Фактичними аргументами можуть бути константи, змінні або вирази.

5. У прототипі функції не обов'язково задавати імена формальних аргументів, достатньо оголосити їх тип, наприклад, у такий спосіб:

```
double seredne(int, int);
```

6. Якщо функція не отримує жодного аргументу, у заголовку функції потрібно ставити порожні дужки (дужки записуються обов'язково) або записати у дужках void:

```
double func();
```

або

```
double func(void);
```

7. Тип значення, яке повертає функція, може бути довільним, але не може бути масивом або функцією. Наприклад, таке оголошення є помилкове:

```
int [10] func();    // Помилка!
```

8. C++ дозволяє існування у програмі функцій з однаковим ім'ям, але різною кількістю аргументів – перевантажувальні функції.

Тепер виконаємо детальний аналіз засобів створення підпрограм мовою C++ з використанням конкретних прикладів. У цій мові підпрограма називається функцією. Програма зазвичай містить головну функцію та кілька допоміжних, які описують розв'язання підзадач основної задачі. Ознайомимося зі створенням функцій на простому прикладі.

Приклад 9.1. Обчислити периметр трикутника на площині, заданого координатами його вершин.

(x1, y1), (x2, y2), (x3, y3) – координати вершин трикутника. Щоб обчислити його периметр, потрібні довжини сторін (відрізків із кінцями у вершинах)

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

$$\sqrt{(x_3 - x_1)^2 + (y_3 - y_1)^2}$$

$$\sqrt{(x_3 - x_2)^2 + (y_3 - y_2)^2}$$

Вони обчислюються однаково, лише з різними парами точок. Однак ці доволі громіздкі й дуже схожі вирази писати тричі не будемо, оскільки всі вони описують конкретне розв'язання такої *загальної підзадачі*: обчислити довжину відрізка за чотирма координатами його кінців.

Довжина залежить від чотирьох величин – *параметрів* підзадачі. Коли розв'язується підзадача, параметри мають конкретні значення-координати – *аргументи* в цьому *конкретному розв'язанні*.

Опишемо розв'язання вказаної підзадачі у вигляді окремої функції з іменем `dist`, параметрами якої є чотири дійсні величини, позначені іменами. У головній же функції напишемо три *виклики* функції `dist`, в яких укажемо координати потрібних нам точок.

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
```

```
#include <cmath>
using namespace std;
double dist(double a1, double b1, double a2, double b2)
// distance from (a1, b1) to (a2, b2)
{
return sqrt((a1-a2)*(a1-a2)+(b1-b2)*(b1-b2));
}
int main()
{
setlocale(LC_ALL, "Ukrainian");
double x1, y1, x2, y2, x3, y3;
cout << "Введіть значення трьох пар координат: ";
// задання конкретних координат
cin >> x1 >> y1 >> x2 >> y2 >> x3 >> y3;
cout << "\nПериметр трикутника = " <<
dist(x1, y1, x2, y2) + //виклик – перша сторона
dist(x1, y1, x3, y3) + //виклик – друга сторона
dist(x2, y2, x3, y3);  //виклик – третя сторона
system("pause");
return 0;
}
```

У поданому програмному ході спочатку означено функцію. Тип значень, які повертаються, указано в її заголовку перед іменем функції. Після імені в круглих дужках оголошено параметри функції – дійсні змінні з іменами `a1`, `b1`, `a2`, `b2`. Після заголовку записано коментар, який хоча й не обов'язковий, але дуже корисний: він повідомляє, що має обчислювати ця функція.

Далі *тіло функції* – блок, який містить послідовність настанов. Тут лише одна настанова, яка задає обчислення і повернення дійсного значення виразу – зарезервоване слово `return` означає "повернути". У головній функції виклики функції `dist` записано у виразі-значенні. Результати викликів функції `dist` повертаються у місце звернення, додаються і ця сума виводиться.

Оголошення параметрів функції, хоча й розташоване за межами блоку функції, діє в цьому блоці до його кінця. Параметри

функції, імена яких оголошено в заголовку, називаються *формальними параметрами*, а вирази або імена змінних, записані у виклику, – *фактичними параметрами*.

Функція може не мати параметрів, але круглі дужки в її заголовку обов'язкові.

Виклик функції є виразом і може виступати операндом в інших виразах. Його типом вважається вказаний у заголовку тип значення, яке повертається.

Довільне виконання функції, яка повертає значення, має закінчуватися виконанням настанови `return` із виразом, тип якого може бути перетворений до типу значень, що повертаються. В іншому разі виконання функції може мати непередбачувані наслідки.

9.3. Передавання параметрів у функцію

Механізм передавання параметрів є основним засобом обміну інформацією між функцією, що викликається, та функцією, яка викликає.

У C++ існує 3 способи передавання аргументів у функцію, а саме:

– передавання аргументів *за значенням* (*Call-By-Value*). Це є простим передаванням копій змінних у функцію. У цьому разі зміна значень аргументів у тілі функції не змінить значення, які передавались у функцію ззовні (при її виклику);

– передавання аргументів *за адресою* змінної. У цьому разі функцію в якості аргументів передаються не копії змінних, а копії адрес змінних, тобто вказівник (*pointer*) на змінну. Використовуючи цей вказівник функція здійснює доступ до потрібних комірок пам'яті, де розташовано значення переданої змінної і може змінювати її значення;

– передавання аргументів *за посиланням* (*Call-By-Reference*). Передається посилання (вказівник) на об'єкт (змінну), що дає змогу синтаксично використовувати це посилання як вказівник і як значення. Зміни, внесені в аргумент, який переданий за посиланням, змінюють початкову копію аргумента викликаючої функції.

9.3.1. Передавання аргументів за значенням

За замовчуванням, якщо аргументом є не масив, застосовується спосіб передавання аргументу за значенням. Для цього створюється копія значення аргументу, яка присвоюється формальному аргументу. Усі операції, виконані всередині функції, стосуються лише копії аргументу і *не впливають на оригінал*, що існує в модулі, який здійснює виклик. Проілюструємо викладенене проєктом програмного коду розв'язання задачі подвоєння значення змінної цілого типу:

```
#include <iostream>
using namespace std;
int twice(int formal)
{
return 2*formal;
}
int main()
{
    setlocale(LC_ALL, "Ukrainian");
int actual, result;
cout << "\nВведіть ціле число actual= ";
cin >> actual;
result = twice(actual);
cout << "\nРезультат роботи функції twice= " <<
result;
system("pause");
return 0;
}
```

Функція `twice()` подвоює свій формальний аргумент `formal`, не змінюючи аргумент `actual`, який після виклику зберігає своє колишнє значення.

9.3.2. Функції, які не повертають значення

У C++ існують функціїї, які після завершення своєї роботи ніяких результатів у викликаючу програму не передають. Розглянемо структуру такої функції:

```
void ім'я_функції(перелік_параметрів)
{
тіло функції;
return;
}
```

`void` – зарезервоване слово, тип даних, який не може зберігати дані. `Void` ніяк більше не використовується і потрібен тільки для того, щоб компілятор міг визначити тип функції. Після слова `void` пишеться назва функції. Одразу за назвою в круглих дужках через кому перелічуються параметри функції, вказуючи їхні типи. Тип необхідно вказувати для кожної змінної окремо.

Якщо функції не передають жодних значень, тоді переліку параметрів може не бути, але круглі дужки обов'язково залишаються. Після заголовку функції в фігурних дужках формується тіло функції. Необов'язкова настанова `return` вказує на завершення функції. За її відсутності функція завершується на закритій фігурній дужці.

Наприклад, розробимо програмний код задачі, яка виводить на кран суму чисел 5 і 3:

Проєкт програмного коду розв'язання задачі:

```
#include <iostream>
using namespace std;
void Sum(int a, int b) // заголовок функції
                      // обчислення суми цілих
```

```
                    // чисел a i b
{
cout << "a + b = " << a + b;
}
int main()
{
Sum(5, 3);      //виклик функції,
                // в якості фактичних
                // параметрів числові константи
return 0;
}
```

9.3.3. Функції, які повертають значення

Розглянемо функції, які після завершення своєї роботи повертають результат. Такі функції можуть повертати значення довільного типу. Розглянемо структуру такої функції:

```
тип_функції ім’я_функції (перелік_параметрів)
{
тіло функції;
return <значення>;
}
```

У заголовку функції необхідно визначити тип даних, який поверне функція, а також після настанови `return` вказати значення, яке буде повертатися. Значення може бути константою, змінною або виразом, але тип цього значення повинен співпадати з типом функції.

Розробимо програмний код розв’язання попередньої задачі, яка виводить на екран суму чисел 5 і 3 за допомогою функції, яка повертає значення.

Проєкт програмного коду розв’язання задачі:

```
#include <iostream>
using namespace std;
// Функція, яка повертає значення суми двох чисел
int Sum(int a, int b)
{
int S = a + b;
return S;
}
int main()
{
setlocale(LC_ALL, "Ukrainian");
cout << "\nСума a + b = " << Sum(3, 5);
// виводимо результат виконання функції
return 0;
}
```

9.4. Приклади програмних кодів з використанням функцій

Приклад 9.2. Обчислити $x = \frac{\sqrt{6}+6}{2} + \frac{\sqrt{13}+13}{2} + \frac{\sqrt{21}+21}{2}$

Усі три доданки заданої формули схожі один на одного, кожен з них можна записати за допомогою спільної формули: $x = \frac{\sqrt{a}+\mathrm{a}}{2}$, де а – число, яке у першому доданку дорівнює 6, у другому – 13, а у третьому – 21. Слушним є створення функції обчислення цієї формули, параметром якої буде а, і тричі викликати цю функцію для кожного з доданків: 6, 13, 21.

Проєкт програмного коду функції та її виклику в основній програмі:

```
#include <iostream>
#include <cmath>
```

```
using namespace std;
double f (double a)
{
return (sqrt(a) + a)/2;
}
int main ()
{ setlocale(0,".1251");
double x = f(6) + f(13) + f(21);
cout<<" Результат x = "<< x << endl;
system ("pause");
return 0;
}
```

Результати роботи:

```
Результат x = 25.3188
```

Приклад 9.3. Обчислити значення виразу $x = \frac{\sqrt{7}+13}{7+\sqrt{13}} + \frac{\sqrt{15}+12}{\sqrt{12}+15} + \frac{\sqrt{21}+32}{\sqrt{32}+21}$.

Формула у цьому прикладі схожа на формулу попереднього завдання і відрізняється лише тим, що функція тут матиме два параметри.

Проєкт програмного коду функції та її виклику в основній програмі:

```
#include <iostream>
#include <cmath>
using namespace std;
double f(double a, double b)
{
return (sqrt(a) + b)/(a + sqrt(b));
}
int main ()
{
setlocale(0,".1251");
```

```
double x = f(7,13) + f(15,12) + f(21,32);
cout << "\nРезультат x = " << x << endl;
system ("pause");
return 0;
}
```

Результати роботи:

```
Результат x = 3.70726
```

Приклад 9.4. Обчислити значення виразу $z = \frac{2\times 5!+3\times 8!}{6!+4!}$

У формулі чотири рази зустрічається обчислення факторіалу, тому, доречно організувати обчислення значення факторіалу у вигляді функції і викликати її, відповідно, чотири рази для різних параметрів.

Проєкт програмного коду функції та її виклику в основній програмі:

```
#include <iostream>
#include <cmath>
using namespace std;
int fact(int n)
{
int c=1;
for(int i=2; i<=n; i++) c*=i;
return c;
}
int main()
{
setlocale(0,".1251");
double                    z                    =
(2.0*fact(5)+3*fact(8))/(fact(6)+fact(4));
cout << "\nРезультат z= " << z << endl;
system ("pause");
return 0;
}
```

Результат роботи:

```
z=162.9032
```

Приклад 9.5. Обчислити суму $f(x) = \sum_{k=1}^{5} \frac{x^{k+1}}{2^k + k}$ трьома варіантами, використовуючи різні настанови циклу:

а) `for`;
б) `while`;
в) `do-while`.

Проєкт програмного коду розроблених функцій та їх виклику в основній програмі:

```
#include <iostream>
#include <cmath>
using namespace std;
double funfor(double x)
{
double S = 0.;
for (int k=1; k<=5; k++)
S += pow(x,k+1)/(pow(2.,k)+k);
return S;
}
double funwhile (double x)
{
double S = 0.;
int k = 1;
while (k <= 5)
{
S += pow(x,k+1)/(pow(2.,k)+k);
k++;
}
return S;
}
double fundowhile (double x)
{
```

```
double S = 0.;
int k = 1;
do
{
S += pow(x,k+1)/(pow(2.,k)+k);
k++;
}
while (k <= 5);
return S;
}
int main ()
{

setlocale(o,".1251");
double x, s1, s2, s3;
cout << " Ввести x = ";
cin >> x;
s1 = funfor(x);
s2 = funwhile(x);
s3 = fundowhile(x);
cout << "for Sum = "<< s1 << endl;
cout << "while Sum = "<< s2 << endl;
cout << "do-while Sum = "<< s3 << endl;
system ("pause");
return 0;
}
Результати роботи:
Ввести x = 2.7
for S = 28.1871
while S = 28.1871
do-while S = 28.1871
```

9.4.1. Опрацювання масивів у функціях

У процесі опрацювання масивів у функціях передавання їх як аргументів завжди здійснюється за адресою, тобто передається адреса першого елемента (початок масиву), а доступ до кожного

з елементів масиву здійснюється як певний зсув (обчислюваний через індекси) від початку масиву.

Передавання одновимірних масивів до функцій як аргументів можна організувати одним зі способів:

```
double funArr (int a[10]);
double funArr (int a[]);
```

За першим способом явно вказано кількість елементів масиву. У другій синтаксичній формі константного виразу у квадратних дужках немає. Оскільки для другого способу інформація про кількість елементів у прототипі відсутня, такі форми припустимі, коли кількість елементів масиву є глобальною змінною або константою. Проте доцільніше передавати розмірність одновимірного масиву до функції окремим параметром, оскільки це зробить таку функцію більш універсальною, адже вона зможе коректно опрацьовувати масиви з різною кількістю елементів:

```
double funArr (int a[], int n);
```

Якщо функція для опрацювання елементів масиву не повинна передавати результат як одне значення, а лише переставляє елементи масиву місцями або змінює їхні значення, тоді таку функцію оголошують з типом результату `void` (нема результату, який повертається).

Оскільки сам масив передається до функції за посиланням, тому довільні змінювання значень елементів масиву у функції буде видно і в основній програмі, яка викликає цю функцію, що проілюструємо таким прикладом.

Приклад 9.6. Створити функцію для обміну місцями максимального та мінімального елементів масиву і перевірити правильність роботи функції для масиву розміром до 12-ти цілих чисел.

Проєкт програмного коду функції та її виклику в основній програмі:

```
#include <iostream>
#include <cmath>
```

```
using namespace std;
void change (int array[], int n)
{
int imin=0, imax=0, tmp;
for (int i=0; i<n; i++)
{
if(array[imin]>array[i]) imin=i;
if(array[imax]<array[i]) imax=i;
}
tmp=array[imin];
array[imin]=array[imax];
array[imax]=tmp;
}
int main ()
{
setlocale(0,".1251");
int a[12],i,n;
cout  <<  "Введіть  кількість  елементів  масиву
n<=12: ";
cin >> n;
if (n>12) n=12;
cout << "\nСформуйте масив a: \n";
for (i=0; i<n; i++)
a[i] = rand() % 6;
for (i=0; i<n; i++)
cout << a[i] << "\t";
cout << endl;
change (a, n);
cout << "\nЗмінений масив a:\n";
for (i=0; i<n; i++)
cout << a[i] << "\t";
cout << endl;
system ("pause");
return 0;
}
```

При передаванні до функції багатовимірних масивів усі розмірності, якщо вони є невідомі на етапі компілювання, мають передаватися як параметри.

Для звичайного статичного двовимірного масиву, коли обидві розмірності є відомі та є константами, заголовок функції матиме вигляд:

```
int sum(int a[4][6])
```

Розглянемо кілька варіантів передавання матриці цілих чисел `a[3][4]` до функції `DrukArr()`, яка здійснює виведення цієї матриці на екран у консолі.

1. Якщо розміри обох індексів є відомі:

```
void DrukArr(int a[3][4])
{
for (int i=0; i<3; i++)
{
for (int j=0; j<4; j++) cout << a[i][j] << "\t";
cout << endl;
}
}
```

Виклик такої функції може бути таким:

```
int x[3][4];
......
DrukArr(x);
```

2. Перша розмірність для обчислення адреси елемента є неважлива, тому її можна передавати як параметр:

```
void DrukArr(int a[][4], int m)
{
for (int i=0; i<m; i++)
{
for (int j=0; j<4; j++) cout << a[i][j] << "\t";
cout << '\n';
```

```
}
}
```

Виклик цієї функції:

```
int x[3][4];
......
DrukArr(x,3);
```

3. Найскладніший випадок – коли треба передавати обидві розмірності. Наведений далі код функції є поширеною помилкою:

```
void DrukArr(int a[][], int m, int n) // Помилка!
{
for (int i=0; i<m; i++)
{
for (int j=0; j<n; j++) cout << A[i][j] << "\t";
cout << '\n';
}
}
```

Зауважимо, що оголошення параметра як `a[][]` є неприпустимим, оскільки для обчислення адреси елемента двовимірного масиву потрібно знати другу розмірність. У такому разі найбільш поширеним і коректним є застосування динамічних масивів, створювання яких буде докладно розглянуто у подальших розділах.

Приклад 9.7. Ввести матрицю дійсних чисел `arr` розмірності (5×5) і за допомогою функції виконати заміну елементів головної діагоналі на середнє арифметичне відповідного рядка.

Проєкт програмного коду функції та її виклику в основній програмі:

```
#include <iostream>
using namespace std;
void arrzamina(double a[5][5])
{
double s;
```

```
for(int i=0; i<5; i++)
{
s=0;
for(int j=0; j<5; j++)
s+=a[i][j];
s/=5;
a[i][i]=s;
}
}
int main ()
{
setlocale(0,".1251");
double arr[5][5];
int i, j;
cout << "\nВведіть матрицю з 5-ти рядків і 5-ти
стовпців:"<<endl;
for(i=0; i<5; i++)
for(j=0; j<5; j++)
cin >> arr[i][j];
for (int i=0; i<5; i++)
{
for (int j=0; j<5; j++)
cout << arr[i][j] << "\t";
cout << '\n';
}
arrzamina(arr);
cout<< "\nПеретворена матриця:" << endl;
for(i=0; i<5; i++)
{ for(j=0; j<5; j++)
cout << arr[i][j] << "\t";
cout << endl;
}
system ("pause");
return 0;
}
```

ВИКОРИСТАНА ЛІТЕРАТУРА

1. Алгоритмізація та програмування: спеціальність 122 "Комп'ютерні науки" / авт. Ю. С. Процик, Т. С. Самотій, М. В. Левкович, кафедра ІТ НЛТУ України. Львів : НЛТУ України, 2017. URL: http://vee.nltu.edu.ua/course/view.php?id=3 – необхідна авторизація.
2. Бєлов Ю. А. Вступ до програмування мовою С++. Організація обчислень: навчальний посібник / Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставровський. К. : Видавничо-поліграфічний центр "Київський університет", 2012. 175 с.
3. Бублик В. В. Об'єктно-орієнтоване програмування : підручник. К. : ІТкнига, 2015. 624 с.
4. Грицюк Ю. І., Рак Т. Є. Програмування мовою С++ : навчальний посібник. Львів : Вид-во Львівського ДУ БЖД, 2011. 292 с.
5. Ментинський С. М., Пелех Я. М. Основи програмування на С++ : навчальний посібник. Львів : Галицька Видавнича Спілка, 2021. 256 с.
6. Ришковець Ю. В., Висоцька В. А. Алгоритмізація та програмування. Частина 1 : навчальний посібник. Львів : Видавництво "Новий Світ-200", 2021. 337 с.
7. Ришковець Ю. В., Висоцька В. А. Алгоритмізація та програмування. Частина 2 : навчальний посібник. Львів : Видавництво "Новий Світ-200", 2021. 315 с.
8. Трофименко О. Г., Прокоп Ю. В., Задерейко О. В. Алгоритмізація та програмування : навчально-методичний посібник. Одеса : Фенікс, 2020. 310 с. URL : http://dspace.onua.edu.ua/handle/11300/12345.
9. Яворський Н. Б., Марікуца У. Б., Андрійчук М. І., Фармага І. В. Лабораторний практикум з дисципліни "Алгоритмізація та програмування" : навчальний посібник. Львів : Видавництво Львівської політехніки, 2018. 191 с.
10. Ярошко С. А. Методи розробки алгоритмів. Програмування мовою С++ : навчальний посібник. Львів : ЛНУ імені Івана Франка, 2022. 248 с.

Інформаційні ресурси

1. http://cpp.dp.ua/lecture/
2. https://www.bestprog.net/uk/sitemap_ua/c/
3. http://cherto4ka.xyz/category/%d0%bf%d1%80%d0%be%d0%b3%d1%80%d0%b0%d0%bc%d1%83%d0%b2%d0%b0%d0%bd%d0%bd%d1%8f-%d1%81/
4. https://ela.kpi.ua/bitstream/123456789/28216/1/Alhorytmizatsiya-ta-prohramuvannia-Praktykum_2019Kublii.pdf
5. http://eadnurt.diit.edu.ua/bitstream/123456789/15729/1/Bandorina.pdf
6. https://nmetau.edu.ua/file/099.pdf

ПРЕДМЕТНИЙ ПОКАЖЧИК

Алгоритм – 7–9, 13–37, 41, 45, 47, 49–51, 55, 65, 86, 121, 125–127, 129–131, 135, 159, 160, 184, 189, 211, 212

Блок-схема – 22, 24–27, 30–36

Вектор – 14, 177, 202–206, 212

Вказівник – 87, 97, 98, 118, 214, 222, 223

Двовимірний масив – 35, 194–196, 223, 234

Декремент – 9, 97, 98, 103, 105, 118, 153

Директива препроцесора – 12, 57, 58, 60, 62, 90–93, 120

Змінна – 22, 29, 76–81, 107–109, 112, 122, 151, 152, 156, 158, 164, 170, 181, 213, 217, 218

Ідентифікатор – 8, 46, 59, 66, 77–80, 97, 103, 117, 178, 214

Ініціалізування – 27, 78, 79, 81, 97, 107, 149, 150, 152, 155, 156, 179, 180, 181, 182, 195–197, 206

Інкремент – 9, 97, 98, 103, 104, 118, 152, 153, 170

Коментарі – 21, 45, 60, 66, 84, 85,133, 221

Константа – 66, 67, 70, 71, 73, 74, 78, 86, 182, 233

Масив – 10, 22, 23, 31–37, 97, 98, 158, 177–183, 185–197, 199, 200, 214, 219, 223, 230–234

Матриця – 35, 199, 204, 205, 208–210, 235

Модифікування – 20, 22, 27, 28, 43, 149, 150, 152, 153, 155

Настанова присвоєння – 9, 82, 102, 106, 108, 109, 193

Настанова умовного переходу – 44, 125–127, 129, 130, 198

Настанова циклу з передумовою – 149, 167, 168

Настанова циклу з постумовою – 30, 149, 172, 173

Одновимірний масив – 35, 177–179, 195, 199, 231

Означення функції – 10, 213, 214–218

Оператор – 8, 9, 11, 66, 75, 76, 80, 82, 83, 90, 93, 96, 97, 100, 101–107, 111, 112, 117, 118, 126, 152, 153, 160, 163, 166, 170, 189

Параметр циклу – 22, 27–29, 31, 35, 37, 150, 151, 153–155, 158, 168, 169

Параметрична настанова циклу – 149–152, 156, 167, 172–174, 176, 183, 197, 229
Підпрограма – 8, 20, 46, 55, 111, 120, 184, 212, 220
Постфіксна форма оператора інкременту – 103–105
Постфіксна формака оператора декременту – 103–105
Потік введення – 9, 111, 112, 115
Потік виведення – 9, 61, 62, 105, 111, 112, 114–116
Префіксна форма оператора декременту – 103–105
Префіксна форма оператора інкременту – 103–105
Присвоєння – 9, 11, 81–83, 100, 102, 103, 106–109, 154, 193
Програмний код – 8, 54, 58, 59, 90, 127, 161, 164, 182, 224, 225
Прототип функції – 10, 52, 59, 184, 213–217, 219, 231
Технологія програмування – 7, 8, 45, 47, 51
Типи даних – 9, 10, 87, 89, 177
Умова – 27, 29, 35, 49, 51, 110, 126, 129, 134, 138, 139, 149, 150–155, 167, 169, 170–173, 176
Функція – 8, 10, 11, 20, 29, 42, 46, 48–51, 53, 55, 57–62, 77, 82, 85–87, 90–94, 97, 111, 117–119, 121, 128, 135–139, 146, 147, 184, 211–234

Авторський колектив

Рудий Тарас Володимирович –
кандидат технічних наук, доцент,
доцент кафедри інформаційного та аналітичного
забезпечення діяльності правоохоронних органів
Львівського державного університету внутрішніх справ,
доцент кафедри електромехатроніки
та комп'ютеризованих електромеханічних систем
Національного університету «Львівська політехніка»
(розділи 6–9);

Паранчук Ярослав Степанович –
доктор технічних наук, професор,
професор кафедри електромехатроніки
та комп'ютеризованих електромеханічних систем
Національного університету «Львівська політехніка»
(розділи 3–5);

Сеник Володимир Васильович –
кандидат технічних наук, доцент,
завідувач кафедри інформаційного
та аналітичного забезпечення діяльності правоохоронних органів
Львівського державного університету внутрішніх справ,
доцент кафедри обчислювальної математики та програмування
Національного університету «Львівська політехніка»
(вступ, розділи 1, 2).

НАВЧАЛЬНЕ ВИДАННЯ

Тарас РУДИЙ
Ярослав ПАРАНЧУК
Володимир СЕНИК

АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

ЧАСТИНА 1

«Структурне програмування»

Навчальний посібник

Редагування *Ірина Краївська*

Макетування *Надія Лесь*

Друк *Іван Хоминець*

Підписано до друку 20.04.2023.
Формат 60×84/16.
Папір офсетний. Умовн.-друк. арк. 13,95.
Зам. № 6-23.

Львівський державний університет внутрішніх справ
вул. Городоцька, 26, Львів, 79007, Україна

Свідоцтво про внесення суб’єкта видавничої справи до державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції
ДК № 2541 від 26 червня 2006 р.