МІНІСТЕРСТВО ОСВІТИ І НАУКИ,
МОЛОДІ ТА СПОРТУ УКРАЇНИ

ДЕРЖАВНИЙ ВИЩИЙ НАВЧАЛЬНИЙ ЗАКЛАД
«НАЦІОНАЛЬНИЙ ГІРНИЧИЙ УНІВЕРСИТЕТ»

В.В. Ткачов, П.Ю. Огєєнко,
Р.В. Макітренко

КОМП'ЮТЕРНІ ТЕХНОЛОГІЇ ТА ПРОГРАМУВАННЯ

Навчальний посібник

Том 1

ТЕОРЕТИЧНІ ВІДОМОСТІ

Дніпропетровськ
НГУ
2012

УДК 004.42 (075.8)
ББК 32.973-018.1я73
Т-48

Рекомендовано Міністерством освіти і науки, молоді та спорту України як навчальний посібник для студентів вищих навчальних закладів напряму підготовки "Автоматизація та комп'ютерно-інтегровані технології" (лист № 1/11-6162 від 03.05.2012 р.).

Рецензенти:

А.І. Купін, д-р техн. наук, професор, завідувач кафедри "Комп'ютерні системи і мережі" (Криворізький технічний університет)

І.В. Жуковицький, д-р техн. наук, професор, завідувач кафедри "Електронні обчислювальні машини" (Дніпропетровський національний університет залізничного транспорту ім. академіка В. Лазаряна)

Ткачов, В.В.

Т-48 Комп'ютерні технології та програмування [Текст]. Т. 1. Теоретичні відомості: навч. посібник / В.В. Ткачов, П.Ю. Огєєнко, Р.В. Макітренко – Д.: Національний гірничий університет, 2011. – 173 с.

ISBN 978-966-350-361-5
978-966-350-362-2 (Т.1)

Розглянуто базисні процедурні особливості за стандартом С89 та об'єктно-орієнтовані підходи розробки програм на мові С++. Наведено лістинги програм для вирішення учбових та прикладних задач.

Посібник укладено відповідно до програми дисципліни "Комп'ютерні технології та програмування" для студентів, що навчаються за напрямом підготовки 050202 "Автоматизація та комп'ютерно-інтегровані технології", а також може бути використано для студентів галузі "Автоматика та управління".

УДК 004.42 (075.8)
ББК 32.973-018.1я73

© В.В. Ткачов, П.Ю. Огєєнко, Р.В. Макітренко, 2011.
© Державний ВНЗ "Національний гірничий університет", 2011.

ISBN 978-966-350-361-5
978-966-350-362-2 (Т.1)

ЗМІСТ

Передмова.. 6
ЧАСТИНА I. ПРОЦЕДУРНЕ ПРОГРАМУВАННЯ.................................... 8
1. ІСТОРІЯ СТВОРЕННЯ І РОЗВИТКУ МОВ С ТА С++.................... 8
1.1. Створення процедурної мови С.. 8
1.2. Створення об'єктно-орієнтованої мови С++............................ 9
1.3. Призначення та основні концепції мов С та С++.................... 9
Контрольні питання.. 12
2. СТРУКТУРА ПРОГРАМ НА МОВІ С++. ОСНОВНІ ПОНЯТТЯ.... 13
2.1. Структура програми.. 13
2.2. Коментування у програмному коді.. 16
2.3. Службові слова мови С++.. 17
2.4. Типи даних.. 17
2.5. Змінні та константи.. 18
2.6. Операції та прості оператори.. 20
2.6.1. Арифметичні операції.. 20
2.6.2. Логічні операції.. 21
2.6.3. Порозрядні операції.. 22
2.6.4. Операції співвідношення... 23
2.6.5. Операції присвоювання... 24
2.6.6. Вплив переприсвоювання на значення змінних......... 24
2.6.7. Пріоритети операцій у виразах.................................. 25
2.7. Локальна та глобальна області видимості змінних................. 26
Контрольні питання.. 27
3. ФОРМАТНИЙ ВВІД/ВИВІД ПРИ РОБОТІ З КОНСОЛЛЮ........... 29
3.1. Використання функцій форматного вводу/виводу................. 29
3.1.1. Функція форматного виводу printf()........................... 29
3.1.2. Функція форматного вводу scanf()............................. 32
Контрольні питання.. 34
4. РОЗГАЛУЖЕННЯ... 36
4.1. Оператори умови... 36
4.1.1. Складений оператор умови if-else............................. 36
4.1.2. Конструкція else if... 43
4.1.3. Оператор вибору switch.. 45
4.1.4. Оголошення змінних у гілках case............................ 47
4.2. Умовна операція "?:".. 48
Контрольні питання.. 49
5. ЦИКЛИ.. 50
5.1. Оператори циклів... 50
5.1.1. Простий оператор циклу з передумовою while.......... 50
5.1.2. Простий оператор циклу з постумовою do-while....... 52
5.1.3. Складний оператор циклу з передумовою for........... 54
5.2. Оператори break та continue... 57
5.3. Оператор goto... 57

Контрольні питання 59
6. ЗМІННІ-ПОКАЖЧИКИ 61
6.1. Доступ до пам'яті за допомогою змінних-покажчиків 61
6.2. Динамічне виділення пам'яті 63
Контрольні питання 64
7. МАСИВИ 65
7.1. Робота з великими об'ємами даних 65
7.2. Масиви 65
7.2.1. Оголошення масиву 66
7.2.2. Робота з елементами одновимірного масиву 68
7.2.3. Робота з елементами багатовимірного масиву 69
7.3. Робота з елементами масивів за допомогою змінних-покажчиків 71
7.4. Динамічні масиви 72
7.5. Застосування масивів 74
Контрольні питання 75
8. СКЛАДЕНІ ТИПИ ДАНИХ 76
8.1. Складені типи даних 76
8.1.1. Структури 76
8.1.2. Поєднання 79
8.1.3. Перелічення 80
Контрольні питання 81
9. ФУНКЦІЇ 83
9.1. Процедурний підхід при написанні програм 83
9.1.1. Оголошення функції 83
9.1.2. Визначення функції 84
9.1.3. Виклик функції 84
9.1.4. Екземпляри структур у якості аргументів функції 86
9.1.5. Посилання та змінні-покажчики у якості аргументів функції 87
9.1.6. Визначення аргументів за умовчуванням 90
9.2. Клас пам'яті змінних функції 91
9.3. Перевантаження функцій 92
9.4. Рекурсія функцій 93
9.5. Застосування функцій 94
Контрольні питання 95
10. ПОВ'ЯЗАНІ СПИСКИ 96
10.1. Пов'язані списки 96
10.1.1. Однозв'язний список 96
10.1.2. Двозв'язний список 100
10.2. Застосування пов'язаних списків 103
Контрольні питання 104
ЧАСТИНА ІІ. ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ 106
11. ПОТОКОВИЙ ВВІД/ВИВІД ПРИ РОБОТІ З КОНСОЛЛЮ 106
11.1. Потоковий ввід/вивід 106

Контрольні питання.. 109
12. ОБ'ЄКТНО-ОРІЄНТОВАНИЙ ПІДХІД. КЛАСИ ТА ЇХ ЧЛЕНИ... 110
12.1. Об'єктно-орієнтоване програмування.................................... 110
12.1.1. Клас.. 110
12.1.2. Доступ до членів класу.. 111
12.1.3. Спеціальні методи класу... 113
Контрольні питання.. 118
13. СПАДКУВАННЯ.. 119
13.1. Спадкування при об'єктно-орієнтованому програмуванні... 119
13.1.1. Створення похідних класів.. 119
13.1.2. Виклик конструктора та деструктора базового класу... 124
13.1.3. Визначення аргументів конструктора базового класу... 125
13.2. Перевантаження методів базового класу у похідному.......... 127
13.3. Множинне спадкування і сутність невизначеності при ньому... 130
Контрольні питання.. 132
14. ВІРТУАЛЬНІ, ДРУЖНІ ТА СТАТИЧНІ ФУНКЦІЇ......................... 134
14.1. Поліморфізм.. 134
14.1.1. Віртуальні функції... 134
14.1.2. Раннє та пізнє зв'язування... 137
14.1.3. Чисті віртуальні функції та абстрактні класи........... 137
14.2. Віртуальний деструктор.. 138
14.3. Віртуальні базові класи... 139
14.4. Дружні функції та дружні класи... 140
14.5. Статичні поля та функції... 142
Контрольні питання.. 146
15. ШАБЛОНИ ФУНКЦІЙ І КЛАСІВ... 147
15.1. Механізм шаблонів... 147
15.1.1. Шаблон функції... 148
15.1.2. Шаблон класу.. 152
Контрольні питання.. 159
16. БАГАТОФАЙЛОВІ ПРОЕКТИ.. 160
16.1. Багатофайлові проекти.. 160
16.2. Міжфайлові змінні... 163
16.3. Міжфайлові функції.. 165
16.4. Міжфайлові класи... 165
16.5. Багатофайлові програми.. 166
Контрольні питання.. 167
Рекомендована література.. 168
Перелік скорочень.. 169
Предметний покажчик.. 170

Присвячується 50-річчю кафедри
автоматизації та
комп'ютерних систем
Державного вищого навчального закладу
"Національний гірничий університет"

Передмова

Впровадження інноваційних методів автоматизації технологічних та виробничих процесів є одним з найважливіших напрямків економічного росту у сфері промисловості та розвитку науки процвітаючої країни.

Спеціалісти у сфері автоматизації повинні у повному обсязі володіти знаннями базових дисциплін: "Теорія автоматичного керування", "Електроніка та мікросхемотехніка", "Мікропроцесорні системи", "Програмування", "Комп'ютерне моделювання процесів і систем", "Електромашинні засоби автоматизації" тощо. Кожна з них є важливою для формування вмінь для проектування, моделювання та розробки автоматизованих систем керування, контролю та моніторингу.

Навчальна дисципліна "Комп'ютерні технології та програмування" для студентів спеціальності 050202 "Автоматизація та комп'ютерно-інтегровані технології" на кафедрі "Автоматизації та комп'ютерних систем" (АКС) викладається у першому та другому семестрах на першому курсі. Задачею предмету є навчання молодих спеціалістів основам алгоритмізації та створення програмних проектів як високого, так і низького рівнів. Базовою для ознайомлення з програмуванням обрано обє'ктно-орієнтовану мову С++, як одну з найбільш поширених та універсальних.

Навчальний посібник складається з двох частин, що включають теоретичні відомості та практичні матеріали для поглибленого вивчення основ процедурного та об'єктно-орієнтованого програмування. Перший том містить дві частини.

Перша частина присвячена процедурному програмуванню на мові С++, яке відповідає стандарту С89. Вона включає теоретичні відомості про принципи побудови програм, розгалуження та зациклення процесу виконання, форматований ввід/вивід при роботі з консоллю, застосування масивів і складених типів даних, роботу з пов'язаними списками та структурування програмного коду на основі створення функцій для вирішення підзадач.

Друга частина розкриває принципи об'єктно-орієнтованих підходів при розробці програмних продуктів. Вона містить огляд властивостей трьох базових понять об'єктно-орієнтованого програмування: інкапсуляції, спадкування та поліморфізму. Теоретичні відомості включають ознайомлення з потоковим вводом/виводом, класами та їх членами, віртуальними, дружніми і статичними функціями та механізмом застосування шаблонів.

Для перевірки засвоєння матеріалу теоретичних частин наприкінці кожен з розділів має перелік контрольних питань. Ключові слова мови С++ у тексті

подано напівжирним шрифтом, а коментарі та пояснення програм – курсивом. Це полегшує сприйняття матеріалу навчального посібника та дозволяє привернути увагу читача до базових конструкцій мови програмування.

Навчальний посібник побудовано у відповідності до програми дисципліни "Комп'ютерні технології та програмування", але його матеріал може бути застосовано і для інших предметів, що базуються на вивченні основ програмування мовою С++.

Автори вважають, що теоретичні відомості та велика кількість практичних прикладів створення програм з детальним їх описом дозволять студентам використовувати посібник для самостійного навчання та полегшать ознайомлення з дисципліною "Комп'ютерні технології та програмування".

ЧАСТИНА І
ПРОЦЕДУРНЕ ПРОГРАМУВАННЯ

1. ІСТОРІЯ СТВОРЕННЯ І РОЗВИТКУ МОВ C ТА C++

Навчальною метою розділу є ознайомлення читача з історичними відомостями створення мов C та C++.

Внаслідок вивчення матеріалу даного розділу читач повинен знати:

– основні дати щодо створення та стандартизації мови C;
– основні дати щодо історії появи та стандартизації мови C++;
– принципову різницю між мовами програмування різних рівнів;
– характеристики мов C та C++;
– концептуальні базові поняття мов C та C++.

1.1. Створення процедурної мови C

Передумови для створення мови програмування, яка дозволила б програмісту розробляти системне програмне забезпечення і при цьому відрізнялася від вже існуючих мов своєю простотою та високим рівнем структуризації, почали формуватися ще в 60-х роках 20-го століття. Великий внесок в розвиток концепції C внесла мова BCPL, створена Мартіном Річардсом. На її основі Кеном Томпсоном була розроблена мова B. Вже після її появи у 1970 році Деннісом Рітчі була створена і реалізована мова, що мала переваги своїх попередниць, а також оригінальний підхід для вилучення їх недоліків. Мова була розроблена для операційної системи Unix на комп'ютері DEC PDP – 11. Вона була названа ім'ям C, що визначає її родову належність до мови B. Протягом багатьох років єдиною версією цієї мови була та, що надходила разом з операційною системою Unix.

Вперше вона описана Брайаном Керніганом та Деннісом Рітчі в книзі "Мова програмування C" (The C Programming Language), яку було видано у 1978 році. Та все ж для стандартизації мови знадобилось ще десять років. У 1983 році було створено комітет, що мав затвердити стандарт ANSI (American National Standards Institute – Національний інститут стандартизації США) для мови C. З цього моменту і до виникнення затвердженого стандарту минуло ще шість років.

Стандарт ANSI було прийнято у грудні 1989 року. Його перша публікація з'явилась вже у 1990 році. Також затверджений стандарт було прийнято ISO (International Standards Organization – Міжнародна організація по стандартизації), тому він називається стандартом ANSI/ISO мови C. На протязі 90-х років до існуючого стандарту було внесено декілька змін, відповідно до яких в C додавалось декілька бібліотечних функцій та механізмів роботи з вузькоспеціалізованими засобами. Таким чином, розвиток мови C зумовив виникнення у 1999 році другого стандарту. Загалом новий стандарт зберіг усі особливості, затвердженні з 1989 року. До нього було додано декілька числових бібліотек, можливість роботи з масивами змінної довжини та інші спеціальні

механізми. Обидва стандарти С до сих пір широко використовуються. Їх прийнято називати С89 та С99 у відповідності до року затвердження.

1.2. Створення об'єктно-орієнтованої мови С++

Мова С++ є розширенням мови С. Вона включає повну версію С, поліпшену за рахунок зміни базової концепції. Мова С є структурною, а мова С++ – об'єктно-орієнтованою.

С++ розроблена Б'єрном Страуструпом у 1979 році у науково-дослідницькому центрі AT&T Bell Laboratories (Нью-Джерсі, США). Спочатку мова отримала назву "С з класами". Цю назву було змінено у 1983 році. Таким чином, робота щодо розширення можливостей С++ ведеться починаючи з 80-х років. За цей час мова перенесла декілька суттєвих модернізацій. У 90-х роках було створено комітет по стандартизації ANSI/ISO мови С++. Мова С з першою поправкою у 1989 році була прийнята у якості базового документу стандарту С++. У січні 1994 року комітетом було представлено перший робочий проект, але затвердження стандартизації було сильно призупинено. Причиною цього стало доповнення до С++ декількох механізмів, що значно розширили її можливості щодо розробки програм. Серед цих механізмів можна виділити стандартну бібліотеку шаблонів, створену Олександром Степановим, і засоби обробки виключних ситуацій під час роботи системних програм.

Більшість розробників С++ компіляторів намагаються підтримувати вже стандартизовану частину, яку прийнято називати Standard С++, та все ж, до сих пір, код, написаний в одному середовищі розробки, часто у своєму вихідному вигляді може бути непереносним на іншу платформу. Таким чином, розширені механізми мови, що є перевагою для програмістів всього світу, які розробляють програмне забезпечення на С++, являють собою також і фактор, що затримує її стандартизацію.

Так на протязі останніх років було видано декілька стандартів мови: С++98 (1998 рік), С++03 (2003 рік), С++TR1 (2007 рік) та С++11 (2011 рік).

Останній варіант стандарту С++ комітетом було опубліковано 12 серпня 2011 року. Його повна назва – "ISO/IEC 14882:2011".

1.3. Призначення та основні концепції мов С та С++

С і С++ відносяться до категорії процедурних мов програмування, оскільки кожний рядок програми в них виступає командою для персонального комп'ютера. Важливо розуміти, що С++ є розширенням мови С або, інакше кажучи, С входить у С++ як підмножина, таким чином, усі особливості стандарту С89 (С99 має деякі особливості, що не увійшли до С++), також характеризують і С++.

С прийнято визначати як мову середнього рівня, що відповідає не думці про неї серед програмістів, а її особливостям застосування.

До мов вищого рівня відносять програмні мови, що дозволяють розробляти програми для користувачів та систем, які звертаються до персонального комп'ютера через програмний прошарок, яким зазвичай виступає ОС. Мови ж низького рівня навпаки забезпечують своїм програмам

безпосередній доступ до апаратного забезпечення, практично не використовуючи засобів для міжпрограмної взаємодії в операційній системі.

C включає у себе як можливості роботи з апаратним забезпеченням, так і засоби створення програм для користувачів, що ставить її серед існуючих мов програмування на середній рівень. Саме ця особливість обумовила те, що в наш час мови C і C++ є одними із самих поширених серед спілки програмістів. Мова C до сих пір продовжує використовуватися та розвиватися. Тому їй ще досить далеко до переходу у розряд "мертвих мов". Основою для цього слугують дві причини: по-перше, навіть зараз, з появою мов C++, Java, Delphi, C# та інших, існує безліч задач, для яких найліпшим засобом вирішення слугує C (ПЗ для мікроконтролерів, прості системні програми, програми для виконання розрахункових задач тощо), а, по-друге, більшість із розроблених раніше C програм до сих пір експлуатуються і вдосконалюються розробниками.

Завдяки особливостям мови програми, що написані за допомогою C, мають високу переносимість, яка дозволяє використовувати їх на різних апаратно-програмних платформах.

Основною структурною конструкцією для C і C++ виступає функція (в деяких інших мовах подібні блоки називають процедурами).

Функція – це сукупність команд, згрупованих під одним ім'ям, що необхідно для уточнення – в якому місці програми ця група має бути застосована. Використання функцій значно спрощує процес розробки програмного забезпечення за рахунок того, що для виділеного таким чином коду, одного разу виправивши, можна гарантувати коректність його роботи незалежно від іншої частини програми.

При цьому функції однієї програми зовсім необов'язково повинні розроблятись одним програмістом. Розробник програми може використовувати функції стандартних бібліотек C, бібліотек, що надходять разом з системою програмування, або бібліотек, що поширюються окремо, через Інтернет або іншими шляхами, і при цьому для нього немає жодної необхідності розбиратися з текстами підпрограм , "прихованими" за їх іменами.

Окрім функцій, в C присутній ще один засіб структуризації – програмний блок. Програмним блоком прийнято називати логічно зв'язану групу команд, які розміщені у фігурних дужках. В C можуть існувати незалежні програмні блоки та програмні блоки, що зв'язані з оператором або функцією. Фактично функція – це також програмний блок, але той, що має певне визначене ім'я.

Будь-яка програма на мові C і C++ містить як мінімум одну структурну одиницю, саме функцію, оскільки незалежні та зв'язані з операторами програмні блоки не можуть існувати за межами функцій.

Програми, що розроблені на основі C, характеризуються високим рівнем модульності, що значно збільшує їх гнучкість порівняно з процедурними мовами. Процедурними прийнято називати мови програмування, в яких відсутні блочні структури (Assembler, Basic). Програмний код таких мов базується на основі формування послідовності команд, що реалізують підзадачі. Для забезпечення розгалуження або зациклювання програм використовуються

команди безумовних переходів. Враховуючи високу структуризацію С, її відносять до структурованих мов.

Окрім сказаного вище, важливо розуміти, що С відноситься до мов, створених спеціально для програмістів. Хоч, як дивно це не звучить, існує безліч мов розроблених не для програмістів, таких як COBOL і Basic.

Основною задачею цих мов є спрощення їх сприйняття і полегшення роботи з ними, а ніяк не прагнення до підвищення гнучкості та універсальності розробленого програмного забезпечення.

На відміну від наведених для прикладу COBOL і Basic, С розроблена і поліпшується програмістами для програмістів. Завдяки цьому вона має невелику кількість обмежень відносно сфер застосування продуктів, що створюються, блочну структуру, великий набір бібліотечних функцій і при цьому невелику кількість ключових слів (слів коду, що визначають складні команди).

Мова С++ в свою чергу пішла далі С. Її прийнято вважати об'єктно-орієнтованою мовою. Основою С++ виступає так званий об'єкт, який більш повною мірою може описувати дані і засоби роботи з ними. Такими засобами і виступають функції. Об'єктна орієнтація мови С++ повинна вирішувати основні недоліки структурного програмування: доступ до глобальних даних абсолютно із усіх частин програми та відокремленість даних і функцій для роботи з ними.

Мова С++ як і С визначає організацію команд у програмі, а не спосіб її виконання на персональному комп'ютері. На комп'ютері програма може бути виконана двома шляхами – інтерпретацією або компіляцією та подальшим запуском виконуючого файлу. Засобом для виконання може слугувати одна із спеціалізованих програм: інтерпретатор або компілятор відповідно.

Інтерпретатор – це програма, яка під час розбору і перевірки написаного програмного коду перетворює та виконує його покомандно або увесь повністю, при цьому не створюючи вихідного файлу.

Компілятор – це програма, яка виконує перевірку коректності написаного програмного коду і перетворює його в програму на машинній мові (Assembler або сукупність двійкових команд). Під час виконання цього процесу створюється об'єктний файл, який і містить еквівалентну вихідній програму на машинній мові.

Після компіляції для створення можливості запуску файлу на виконання процесором персонального комп'ютера може використовуватись іще одна спеціальна програма – ліннер (часто вбудований у компілятор). Результатом її роботи є так званий файл, що виконується (executable), який має розширення ".exe". Він є автономним додатком, який може безпосередньо взаємодіяти з операційною системою для виконання заданої задачі.

Компіляція і інтерпретація, приводять до аналогічного результату – виконанню програми, але при цьому, якщо при компіляції перевірка і перетворення початкової програми в команди процесору відбувається один раз, то при інтерпретації цей процес повторюється під час кожного запуску. Таким чином, для персонального комп'ютера компіляція програми є більш легкою задачею у порівнянні з інтерпретацією.

Мови програмування зазвичай розробляються з урахуванням оптимізації для одного із процесів: компіляції або інтерпретації. Мова Java, наприклад, розрахована на інтерпретацію, а С і С++ на компіляцію.

Мови С і С++ дуже сильно залежать від своїх компіляторів. Вони здатні розширювати або звужувати специфіку мов за рахунок жорсткої прив'язки до системи програмування, з якою надходять.

Компілятор С++ майже завжди здатен обробити і програму, написану на С, за виключенням програм, які містять елементи розширення С, що входять у стандарт С99. Та все ж, компілятор, що призначається для компіляції С програм, просто не зможе зрозуміти код програми на С++. Для відмінності програм мов С і С++ прийнято використовувати різні розширення для їх файлів – ".c" та ".cpp" відповідно.

Висновки

У даному розділи було розглянуто наступні основні питання:

– історія створення процедурної мови С;

– історія створення об'єктно-орієнтованої мови С++.

Контрольні питання

1) Яка мова програмування була основою для створення С?
2) Хто розробив концепції мови С?
3) У якому році була опублікована перша книжка про мову С?
4) Коли було створено комітет стандартизації мови С?
5) Яка значна подія для мови С сталася у 1989 році?
6) Скільки є стандартів мови С і як вони називаються?
7) Коли з'явилася перша публікація стандарту С?
8) Хто є розробником мови С++?
9) Яку назву на початку мала мова С++?
10) У якому році було створено мову С++?
11) Коли було створено комітет стандартизації мови С++?
12) Яка значна подія для мови С++ сталася у 1989 році?
13) У чому причина призупинення стандартизації мови С++?
14) На які рівні поділяють мови програмування?
15) Що називають програмним блоком і які вони бувають?
16) Яке значення має вислів "мова програмування для програмістів"?
17) Чому мову С++ прийнято називати об'єктно-орієнтованою?
18) Дайте визначення поняттю інтерпретатор.
19) Чи можна сказати, що мови С та С++ є повністю сумісними та взаємозамінними?
20) Які розширення мають файли програм мов С та С++?

2. СТРУКТУРА ПРОГРАМ НА МОВІ С++. ОСНОВНІ ПОНЯТТЯ

Навчальною метою розділу є ознайомлення читача зі структурою та основними поняттями програм, що розробляються на мові С++.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– створювати прості програми на мові С++;
– визначати необхідний тип даних для створення змінної;
– виконувати оголошення змінних;
– застосовувати змінні для збереження значень та розрахунків математичних і логічних виразів;
– коментувати та форматувати програмний код;
– використовувати наступні види операторів: арифметичні, логічні, порозрядні, співвідношення та присвоювання;
– формувати вирази, враховуючи пріоритети операцій;
– відрізняти глобальні та локальні змінні.

2.1. Структура програми

Під час написання програмного коду на мові С++ необхідно дотримуватись певної структури, оскільки, хоч семантика цих мов і допускає безліч вільностей, від порядку, в якому записуються програмні блоки, залежить чи будуть вони коректно (як задумано програмістом) зв'язані між собою в єдину функціонуючу послідовність.

Таким чином, структура програми задає порядок, відповідно до якого програміст повинен підключати бібліотеки, оголошувати та визначати змінні функції, структури і класи, формувати залежні між собою програмні блоки.

Програмний код повинен бути написаний таким чином, щоб його умовно можна було розбити на наступні рівні:

1) директиви препроцесора
2) оголошення глобальних змінних, функцій, структур і класів
3) визначення глобальних функцій
4) точка входу в програму (головна функція)
5) тіло основної програми
6) визначення глобальних функцій та методів класів

Наступні визначення дозволять більш детально зрозуміти перший рівень:

Директиви – це спеціальні команди, дію яких направлено на компілятор або на комп'ютер.

Препроцесор – спеціальна частина системи програмування, що використовується в якості компілятора. Вона виконується безпосередньо перед компіляцією програмного коду.

Компілятор – програма, до завдання якої входить перевірка коректності програмного коду, написаного програмістом, на відповідність правилам лексики, синтаксису та семантики мови, перетворення цього коду у машинну мову (код, що зрозумілий процесору персонального комп'ютера) та створення об'єктного файлу, котрий і зберігає вихідну версію програми на машинній мові.

Компіляція – процес перетворення початкового програмного коду в машинну мову.

Директиви препроцесора – це команди, що призначені компілятору для виконання дій, які необхідно провести до виконання компіляції програмного коду.

Таким чином, на першому рівні програми на мові С++ містяться команди, що мають бути виконані компілятором до початку компіляції.

На другому рівні розміщуються команди, які повідомляють процесору комп'ютера про те, які в межах програми будуть існувати дані, структури даних та функції роботи з даними.

Третій рівень включає програмні блоки, виконання яких асоціюється з оголошеними раніше функціями.

Четвертий та п'ятий рівні нерозривно зв'язані та являють собою ім'я та тіло програмного блока, з якого буде починатися виконання програми. Такий програмний блок в програмі може бути лише в одному екземплярі та повинен відповідати певним правилам, тому він найчастіше називається точкою входу у програму.

Шостий рівень включає в себе програмні блоки для тих функцій, які не містились на третьому рівні.

Дотримуючись вище описаної структури, програміст не лише підвищує читаність своєї програми, але і полегшує завдання компілятора – перевірку та перетворення коду. Також при дотриманні структури зменшується час виправлення програми, оскільки програміст обходить помилки взаємозв'язку програмних блоків.

Приклад простої програми на мові С++ наведено нижче. Дана програма виводить на консоль (засіб вводу/виводу даних зазвичай зв'язаний з програмами, які не мають власних вікон) привітання користувача.

```
//1-й рівень – директиви препроцесора
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

//4-й рівень – точка входу в програму
void main()                           //оголошення та визначення головної функції
{
//5-й рівень – тіло основної програми
        printf ( "Hello, my dear friend!" );      //вивід константного рядка
}
```

Наведена програма містить лише три з вище перелічених шести рівнів: перший, четвертий і п'ятий. У програмах на мові С++ важливим є не наявність всіх шести рівнів описаної структури, а дотримання їх порядку під час написання програмних блоків.

Перший рядок програми відноситься до першого рівня структури і містить директиву include, яка починається з символу "#".

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу
```

Це говорить про те, що дана директива – команда препроцесора. **#include** – директива процесора, що найбільш часто використовується і слугує для підключення заголовного файлу, як системи програмування так і користувача. Ім'я файлу, що підключено в такий спосіб, відповідно до формату використання директиви розміщується в кутових дужках або лапках. Таких файлів існує досить багато. Вони містять підготовлені у вигляді функцій програмні блоки, які можуть використовуватись для виконання дій над даними в програмі, що розробляється. Під час розгляду особливостей мови С++ ми поступово ознайомимося з заголовними файлами, що найбільш часто використовуються.

Заголовні файли відповідно до мови С повинні мати розширення "**.**h". Для мови С++ допускається їх запис без розширення. Як було сказано раніше, ім'я заголовного файлу може міститися в кутових дужках

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу
```

або в подвійних лапках

```
#include "header.h"                   //підключення заголовного файлу
```

Використання того чи іншого способу запису визначає, в якій директорії на жорсткому диску буде проведено пошук заголовного файлу, що підключається. Якщо використовуються кутові дужки, тоді даний файл імовірно є частиною системи програмування і розташовується в чітко визначеній директорії (найчастіше вона називається "INCLUDE"). При використанні подвійних лапок пошук проводиться в робочому каталозі (найчастіше це директорія, де розміщено виконуючий файл програми, що розробляється). Підключені за допомогою подвійних лапок заголовні файли зазвичай розробляються самим програмістом.

Заголовний файл stdio.h містить функції для здійснення операцій вводу/виводу, про що повідомляють останні дві букви його імені io (і – input (ввід), o – output (вивід)). У наведеному вище прикладі цей файл підключається для того, щоб стало можливим використання функції printf().

Другий рядок програмного коду відноситься до четвертого рівня структури і є точкою входу у програму.

```
void main()                           //оголошення та визначення головної функції
```

Це назва програмного блока, що розташовано за даним рядком. В цьому випадку фігурує функція main(), яка в програмах, написаних на мові С++, є точкою входу, тобто програмним блоком, з якого починається виконання програмного коду. Ім'я main обрано не випадково – в перекладі з англійської воно має значення головний(а), з огляду на блок або функцію.

П'ятий рівень у прикладі подано програмним блоком, що є тілом основної програми. Це іменований програмний блок, що пов'язано з головною функцією. Четвертий та п'ятий рівні, як вже зазначалося, є нерозривними, бо характеризують єдиний структурний елемент програми.

Згідно із синтаксисом C++ всі інструкції програми (вирази, виклики функції тощо), що не є директивами препроцесора та блоками, розділяються за допомогою символа завершення ";". Втрата цих символів є найбільш поширеною помилкою при написанні програм.

Інші рівні буде поступово розглянуто впродовж знайомства з мовою C++.

Треба зазначити, що наведений вище приклад програми написано згідно з правилами форматування. Так прийнято називати особливості подання коду, завдячуючи яким він набуває підвищеної читаності. Відповідно до цих правил структурні блоки програми відокремлюються порожнім рядком та кожний код, що вкладено до блоку, зміщується на одну табуляцію чи п'ять пробілів. Такий підхід дозволяє візуально подати програму найбільш структурованою. Окрім того, правильне форматування полегшує коментування.

2.2. Коментування у програмному коді

Коментування є дуже важливим при написанні програмного коду. Добре відомо, що програми, які не було прокоментовано, з часом стають важкими для сприйняття навіть тим програмістом, який приймав пряму участь в їх створенні. Окрім того, спроможність коментування коду надає розробнику можливість поступового відлагодження програмних фрагментів, забезпечуючи відсутність впливу блоків, що закоментовано.

Таким чином, коментування сприяє як сприйняттю програми, так і її коректному написанню.

У мові C є дві форми коментувань: для одного рядка і для фрагмента тексту. Формат використання коментування наступний:

// *<рядок, що закоментовано >*
/* *<текст, що закоментовано >**/

Коментування виступає частиною програми під час її написання програмістом, але повністю видаляється компілятором при створенні файлу, що виконується. Більшість компіляторів, з якими працює програміст, мають функцію виділення коментувань окремим кольором.

Треба зазначити, що форма коментування фрагмента тексту є більш універсальною, бо коментується тільки той текст, що потрапляє між знаками "/*", "*/". Завдячуючи цьому, коментувати можна як частину виразу, так і програмний блок. Другий формат коментує лише один рядок, причому весь, починаючи зі знаку "//".

Надалі кожен наведений програмний приклад буде мати коментування для полегшення сприйняття лістингів читачем.

При написанні програм можуть бути використані два типи коментування: коментування для кожного рядка коду та коментування для програмних блоків.

Програмістам-початківцям краще використовувати перший тип, бо так можна більш детально розібратися з матеріалом, що вивчається. Взагалі, коментування не є невід'ємною частиною програми, більш того, відсутність коментування ніяк не впливає на програмний код. Коментування – це вільний опис коду, що додається програмістом під час розробки для підвищення легкості подальшого сприйняття програми.

2.3. Службові слова мови C++

Службовими чи ключовими словами у мові програмування прийнято називати слова чи скорочення слів, що для компілятора визначають команду. Відповідно до цього слова не припустимо використовувати для спроб іменування даних.

Згідно зі стандартом C89 у мові C++ зазначені службові слова:

auto	**do**	**goto**	**signed**	**unsigned**
break	**double**	**if**	**sizeof**	**void**
case	**else**	**int**	**static**	**volatile**
char	**enum**	**long**	**struct**	**while**
const	**extern**	**register**	**switch**	
continue	**float**	**return**	**typedef**	
default	**for**	**short**	**union**	

Наступні слова не входили до мови C, вони є командами, що було введено безпосередньо у C++:

asm	**delete**	**operator**	**public**	**throw**
bool	**friend**	**private**	**template**	**try**
catch	**inline**	**protected**	**this**	**virtual**
class	**new**			

Більшість з наведених службових слів буде розглянуто під час подальшого ознайомлення з особливостями написання програм на мові C++.

2.4. Типи даних

Типом даних у мові програмування називають службове слово, що зазначає характеристику значення змінної та ймовірні дії з ним.

Основними типами даних мови C++ є наведені в табл. 2.1.

Таблиця 2.1

Основні типи даних

Назва	Призначення
bool	логічний тип даних
char	символьний, цілочисельний тип даних
int	цілочисельний тип даних
float	тип даних з плаваючою точкою
double	тип даних з плаваючою точкою, подвійної довжини
void	порожній тип даних

Застосування службових слів, наведених у табл. 2.2, дозволяє зробити використання типів даних більш гнучким.

Таблиця 2.2

Службові слова для поширення можливостей типів даних

Назва	Призначення
signed	позитивний діапазон значень
unsigned	діапазон має від'ємні та позитивні значення
short	скорочення діапазону в два рази
long	розширення діапазону в два рази

Згідно з наведеними вище службовими словами у мові С++ можна вказувати типи даних, зазначені у табл. 2.3.

Таблиця 2.3

Повний перелік типів даних

Назва	Розмір у байтах (бітах)	Інтервал зміни значень
bool	1 (8)	від 0 до 1
char	1 (8)	від -128 до 127 чи таблиця ASCII
unsigned char	1 (8)	від 0 до 255 чи таблиця ASCII
signed char	1 (8)	від -128 до 127 чи таблиця ASCII
int	2 (16)	від -32768 до 32767
unsigned int	2 (16)	від 0 до 65535
signed int	2 (16)	від -32768 до 32767
short int	2 (16)	від -32768 до 32767
unsigned short int	2 (16)	від 0 до 65535
signed short int	2 (16)	від -32768 до 32767
long int	4 (32)	від -2147483648 до 2147483647
signed long int	4 (32)	від -2147483648 до 2147483647
unsigned long int	4 (32)	від 0 до 4294967295
float	4 (32)	від 3.4Е-38 до 3.4Е+38
double	8 (64)	від 1.7Е -308 до 1.7Е +308
long double	10 (80)	від 3.4Е-4932 до 3.4Е+4932

Тип даних **void** використовується лише для функцій та змінних-покажчиків.

2.5. Змінні та константи

Змінна – це ім'я, що пов'язано з однією чи декількома послідовними комірками пам'яті, які може бути застосовано для збереження даних. Вона використовується для тимчасового зберігання та оперування статичними та динамічними даними програми. Через змінну реалізується доступ читання та запису пам'яті, що з нею пов'язано.

Рядок у програмному коді, що визначає створення нової змінної та задає її тип даних, називають оголошенням. Формат оголошення наведено нижче:

<*тип даних*> <*список імен змінних*>;

Під час оголошення може бути створено декілька змінних. Їх імена в такому випадку записують через оператор ",".

Згідно із синтаксисом мови програмування імена змінних можуть містити латинські букви, цифри та символ нижнього підкреслювання, але цифра не може бути першою буквою. Зазвичай імена обирають у відповідності до змісту даних, для збереження яких застосовано змінну.

Наприклад:

```
int num;            //оголошення цілочисельної змінної для номера
int x, y, z;        //оголошення цілочисельних змінних для координат
float rval;   //оголошення змінної з плаваючою точкою для дійсного значення
```

Запис значення у змінну виконується через її визначення. Цю дію також прийнято називати ініціалізацією змінної. Визначення може бути виконане як при оголошенні, так і окремо у програмному коді, але обов'язково після оголошення. Для запису значення у змінну використовується операція присвоювання "=".

Наприклад:

```
int x=7, y, z;//оголошення цілочисельних змінних та визначення абсциси точки
y=10;               //визначення ординати точки
z=25;               //визначення аплікати точки
```

Константними вважаються такі змінні, що були оголошені за допомогою службового слова **const**. Вони також є іменами, що пов'язані з пам'яттю, але на відміну від звичайних змінних константні можна визначити впродовж програми лише одного разу. Найчастіше вони ініціалізуються при оголошенні. Застосування константних змінних є рідким, бо необхідності їх впровадження у програми немає. При уважному написанні коду звичайні змінні можуть грати ту саму роль.

Формат оголошення та визначення константної змінної наведено нижче:

const <*тип даних*> <*ім'я змінної*>=<*значення*>;

Наприклад, для програми може бути зазначено число математичної константи ПІ:

```
const float pi=3.14159;         //оголошення та визначення константи ПІ
```

Оперування змінними пов'язано з їх використанням для розрахунків різноманітних математичних та логічних виразів.

2.6. Операції та прості оператори

Майже всі програми, що розробляються програмістами, побудовано на базі використання виразів. У свою чергу вирази створюються на основі операцій. Таким чином, можна сміливо стверджувати, що операція є базисом при програмуванні.

Операцією у мові С++ називають дію, що виконується над даними за допомогою певного простого оператора.

Існує п'ять основних груп операцій: арифметичні, логічні, порозрядні, співвідношення та присвоювання. Окрім того, всі операції поділяються на два типи: унарні та бінарні. Унарні виконуються над однією змінною чи значенням, а бінарні з двома.

2.6.1. Арифметичні операції

До арифметичних операцій відносять операції, що наведено у табл. 2.4.

Таблиця 2.4

Арифметичні оператори

Оператор	Операція
-	віднімання, а також унарний мінус
+	додавання
*	множення
/	ділення
%	ділення за модулем
--	декрементація
++	інкрементація

Оператори *, /, +, – є добре відомими. Їх використання для побудови операції майже нічим не відрізняється від розглянутого у математиці. Особливість є лише у ділення.

Вирази "1/5" та "1.0/5" будуть мати різний результат. 0 та 0.2, бо при використанні з оператором "/" лише цілих значень результат теж зводиться до цілого. Операція ділення за модулем, що заснована на операторі "%", дозволяє знаходити залишок від ділення за наступним принципом:

C=N*A + B, C/A=N + B/A

Результатом операції є наведене вище значення B. Таким чином, при розрахунку 5%2 та 15%6 буде відповідно 1 та 3. Ділення за модулем виконується тільки для цілих змінних. Спроба задіяти значення з крапкою, що плаває, призведе до помилки.

Операції інкремента та декремента нові для сприйняття. Вони є унарними і застосовуються лише для змінних. Використання інкремента чи декремента з константними значеннями викличе помилку.

Інкрементування змінної збільшує, а декрементування відповідно зменшує її значення на одиницю. Обидві операції можуть бути застосовані у

двох формах: префіксній, коли оператор записується перед змінною, та постфіксній – оператор за змінною. Форма запису впливає на значення, що буде узято для розрахунку виразу, в якому задіяна змінна. Так, для

a=b++

та

a=++b

при початковому значенні b=7, результати будуть 7 та 8 при тому, що значення b в обох випадках стане дорівнювати 8. При застосуванні постфікстної форми для розрахунку береться поточне значення змінної і потім виконується її збільшення чи зменшення. Префіксна ж форма застосування дозволяє спочатку здійснити саму операцію, а вже після цього задіяти змінну.

2.6.2. Логічні операції

Логічні операції відповідають бінарній логіці і при виконанні результатом мають одне з двох значень: 0 чи 1.

Мова C++ включає три логічних операції, які наведено у табл. 2.5.

Таблиця 2.5

Логічні оператори

Оператор	**Операція**
&&	І
||	АБО
!	НЕ

Операції "І" та "АБО" є бінарними, а "НЕ" унарною.

Особливості застосування логічних операцій засновані на таблицях істинності (табл. 2.6).

Таблиця 2.6

Таблиці істинності логічних операторів

"І"

x	**y**	**z**
0	0	0
0	1	0
1	0	0
1	1	1

"АБО"

x	**y**	**z**
0	0	0
0	1	1
1	0	1
1	1	1

"НЕ"

x	**z**
0	1
1	0

Згідно з тим, що результатом логічної операції можу бути лише "0" чи "1", то і значення, над якими операція виконується, розглядаються у програмі відповідно. Будь-яке значення, що не є "0", вважається за "1".

Таким чином, для розуміння логічних операцій є достатнім вивчити таблиці істинності.

Наприклад, наступні вирази у програмі буде пораховано, як вказано нижче:

1 && 0 = 0
0 || 1 = 1
при A=1, !A=0
15 && 1 = 1
143 || 0 = 1

2.6.3. Порозрядні операції

Порозрядні операції (табл. 2.7) застосовуються в основному з цілими значеннями. Їх назва походить від особливості дії – обробці підлягають окремо всі розряди значень.

Таблиця 2.7

Порозрядні операції

Оператор	**Операція**
&	І
|	АБО
^	виключне АБО
~	доповнення до одиниці (НЕ)
>>	зсув вправо
<<	зсув вліво

Операції "І", "АБО" та "НЕ" мають схожий принцип дії з вже розглянутими однойменними логічними операціями. Різниця в тому, що таблиці істинності застосовуються окремо до кожної пари розрядів чисел.

Операція "Виключне АБО" також має свою таблицю істинності (табл. 2.8).

Таблиця 2.8

Таблиці істинності порозрядних операторів

"І"

x	**y**	**z**
0	0	0
0	1	0
1	0	0
1	1	1

"АБО"

x	**y**	**z**
0	0	0
0	1	1
1	0	1
1	1	1

"НЕ"

x	**z**
0	1
1	0

"Виключне АБО"

x	**y**	**z**
0	0	0
0	1	1
1	0	1
1	1	0

Використання цих операцій треба розглядати у двійковій формі. Наприклад, 17^25 та 113&10 вирішуються, як зазначено нижче:

^	0	0	0	1	0	0	0	1	(17)
	0	0	0	1	1	0	0	1	(25)
	0	0	0	0	1	0	0	0	(8)

&	0	1	1	1	0	0	0	1	(113)
	0	0	0	0	1	0	1	0	(10)
	0	0	0	0	0	0	0	0	(0)

Операції "І", "АБО" та "Виключне АБО" є бінарними, а "НЕ", "зсув вліво" та "зсув вправо" – унарними.

Але на відміну від "НЕ" операції зсуву потребують допоміжного значення для уточнення. При виконанні цих операцій поточне значення розрядів зсувається на вказане число позицій. З протилежного до направлення зсуву боку значення доповнюється нулями.

Таким чином, при зсуві вліво на три розряди значення 15 та вправо на 2 розряди значення 136 вирішення буде мати наступний вигляд:

15<<3
0 0 0.0 0 1 1 1 (15) => 0 0 1 1 1 0 0 0 (56)
←
136>>2
1 0 0 0 1 0.0 0 (136) => 0 0 1 0 0 0 1 0 (34)
→

2.6.4. Операції співвідношення

Операції співвідношення, як і арифметичні, добре відомі з математики. Їх опис наведено у табл. 2.9.

Таблиця 2.9

Оператори співвідношення

Оператор	Операція
>	більше
>=	більше чи дорівнює
<	менше
<=	менше чи дорівнює
!=	не дорівнює

Широке застосування такі операції знаходять у відтворенні програмних розгалужень, бо відповідають побудові тотожностей, що можуть бути правильними чи неправильними (виконуватись чи не виконуватись). Таким чином, результат операції співвідношення збігається з результатом логічної операції – це "0" або "1". Але на відміну від логічних, операції співвідношення

застосовують до будь-яких чисел. При вирішенні виразів 15<147 та 113==113.2 відповіді будуть 1 та 0.

2.6.5. Операції присвоювання

Остання група операцій є найпоширенішою, бо застосовується для визначення змінних у програмі. Це операції присвоювання (табл. 2.10).

Таблиця 2.10

Оператори присвоювання

Оператор	Операція
=	присвоювання
*=	множення з присвоюванням
/=	ділення з присвоюванням
%=	залишок від ділення з присвоюванням
– =	віднімання з присвоюванням
+=	додавання з присвоюванням
<<=	зсув вліво з присвоюванням
>>=	зсув вправо з присвоюванням
&=	порозрядне І з присвоюванням
|=	порозрядне АБО з присвоюванням
^=	порозрядне виключне АБО с присвоюванням
~=	порозрядне НЕ з присвоюванням

Найчастіше використовується просте присвоювання "=", що дозволяє виконати запис значення, що розраховано у правій частині виразу у змінну, що зазначена зліва.

Наприклад, а=21 говорить про те, що змінну а буде визначено за допомогою значення 21.

Складні форми присвоювань є поєднанням операції присвоювання з іншою операцією і використовуються лише тоді, коли вказане праворуч значення повинно здійснити вплив на змінну з подальшим зберіганням результату. Наприклад:

вирази:	можна замінити на відповідні:
A=A+15	A+=15
B=B*10	B*=10
C=C%2	C%=2
D=D&&25	D&&=25
E=E^43	E^=43

2.6.6. Вплив переприсвоювання на значення змінних

Виконання переприсвоювання значень між змінними, що мають різний тип даних, може призвести до спотворення даних. Тому при роботі зі змінними потрібно чітко уявляти їх призначення і уважно застосовувати операції присвоювання. Ймовірні втрати наведені у табл. 2.11.

Таблиця 2.11

Правила перетворення типів даних

Результуючий тип	**Тип виразу**	**Ймовірні втрати**
signed char	**char**	якщо значення більше за 125, результатом буде від'ємне число
char	**short int**	старші 8 бітів
char	**int** (16 бітів)	старші 8 бітів
char	**int** (32 біти)	старші 24 біти
char	**long int**	старші 24 біти
short int	**int** (16 бітів)	немає
short int	**int** (32 бітів)	старші 16 бітів
int (16 біт)	**long int**	старші 16 бітів
int (32 біти)	**long int**	немає
int	**float**	дробова частина та, можливо, щось ще
float	**double**	точність, результат округляється
double	**long double**	точність, результат округляється

2.6.7. Пріоритети операцій у виразах

При розрахунку виразів, що включають більш ніж одну операцію, компілятор застосовує правило пріоритету. Згідно з ним кожну операцію буде виконано в запланованому порядку. Тому для коректної побудови логіки програми кожен програміст повинен знати пріоритети операцій. Табл. 2.12 відображає всі операції мови C++ у порядку спадання їх пріоритету.

Таблиця 2.12

Операції C++ у порядку спадання пріоритету

<table>
<tr><td>найвищий</td><td>() [] -></td></tr>
<tr><td></td><td>! – ++ -– + – (тип) * & sizeof</td></tr>
<tr><td></td><td>* / %</td></tr>
<tr><td></td><td>+ -</td></tr>
<tr><td></td><td><< >></td></tr>
<tr><td></td><td>< <= > >=</td></tr>
<tr><td></td><td>= = !=</td></tr>
<tr><td></td><td>&</td></tr>
<tr><td></td><td>^</td></tr>
<tr><td></td><td>|</td></tr>
<tr><td></td><td>&&</td></tr>
<tr><td></td><td>||</td></tr>
<tr><td></td><td>?:</td></tr>
<tr><td></td><td>= += -= *= /= і т.д.</td></tr>
<tr><td>найнижчий</td><td>,</td></tr>
</table>

Згідно із наведеною таблицею вирази:

a=110+21^56>>2&&40
b=!17*~36<<4<=5

будуть вирішуватись за наступними алгоритмами:

a=110+21^56>>2&&40
a=131^56>>2&&40
a=131^14&&40
a=141&&40
a=1

b=!17*~36<<4<=5
b=0*~36<<4<=5
b=0*219<<4<=5
b=0<<4<=5
b=0<=5
b=1

2.7. Локальна та глобальна області видимості змінних

Кожна змінна у програмі може бути оголошена як у програмному блоці, так і незалежно від нього. Згідно з цим розрізняють локальну та глобальну області видимості. Якщо оголошення виконано у програмному блоці, то змінну відносять до локальної видимості. Це означає, що змінна існує лише у цьому блоці і при досягненні закриваючої фігурної скобки виділену під зберігання значення пам'ять буде вивільнено. Особливістю локальних змінних є те, що незалежно від того, які дії буде виконано над ними, у програмному блоці це ні в якому разі не зможе завдати впливу на зовнішню частину програми.

На відміну від локальних, оголошення глобальних змінних виконують зовні будь-яких програмних блоків. Такі змінні можуть буди використані будь-де у програмі, зважаючи лише на те, що застосування змінних може мати місце тільки після її оголошення.

Глобальні змінні відкривають шляхи взаємодії між різними програмними блоками, але, використовуючи їх, програміст повинен бути обережним, бо вплив на такі змінні у багатьох програмних фрагментах може стати причиною виникнення логічних помилок.

Наступний приклад демонструє особливості застосування глобальних та локальних змінних.

```
#include <stdio.h>                  //підключення бібліотеки вводу/виводу

int A;                              //оголошення глобальної змінної

void main()                         //оголошення та визначення головної функції
{
        A=7;                        //визначення глобальної змінної
```

```
	printf("A=%i \r\n", A);		//вивід значення на консоль
	{
		int B;			//оголошення змінної
		B=15;			//визначення змінної
		printf("B=%i \r\n", B);	//вивід значення на консоль
		A+=B;			//розрахунок суми
	}
	printf("A=%i \r\n", A);		//вивід значення на консоль
	//printf("B=%i \r\n", B);		//викличе помилку
}
```

У програмі оголошено глобальну змінну A*, якій на початку головної функції присвоюється значення* 7*. Після цього поточне значення* A *виводиться на консоль. Далі створено незалежний неіменований програмний блок, в якому виконано оголошення змінної* B *і присвоєно їй значення* 15*. За допомогою функції* printf() *це значення виводиться на консоль, а потім вміст* B *додається до* A*. Наприкінці програми, після закриття блоку, значення* A *знову виводиться на консоль. Останній рядок закоментовано, бо його виконання призвело б до появи помилки компілятора. Це пов'язано з тим, що змінна* B *у цій точці програми більше не існує.*

Результатом виконання програми будуть рядки:

```
A=7
B=15
A=22
```

Область видимості може бути визначено не тільки для змінних, але й для інших програмних елементів, таких як структури, функції, класи тощо.

Висновки

У даному розділи було розглянуто наступні основні питання:
– структура програми на мові C++;
– коментування програм;
– основні типи даних мови C++;
– змінні та константи;
– операції та оператори мови C++.

Контрольні питання

1) З яких основних рівнів складається програма на мові C++?
2) Дайте визначення компілятора.
3) Що називають директивами препроцесора?
4) Що таке точка входу в програму?
5) У чому полягає сенс застосування кутових дужок та подвійних лапок при підключенні заголовного файлу?

6) Яким символом згідно із синтаксисом мови С++ повинно бути закінчено кожну інструкцію?

7) Які є види коментування?

8) Що називають ключовим словом мови програмування?

9) Які типи даних є для мови С++ основними?

10) Яке службове слово зменшує діапазон значень типу даних в два рази?

11) Для чого застосовується службове слово **unsigned**?

12) Дайте визначення поняттю змінна.

13) Як від звичайних відрізняються константні змінні?

14) У чому сенс оголошення та визначення змінної?

15) Які є групи операцій над змінними?

16) Яке значення є результатом операцій співвідношення?

17) У чому особливість застосування оператора ділення за модулем?

18) Чим відрізняються глобальна та локальна області видимості змінних?

19) Який вплив створює перепривсоювання на значення змінних?

20) У якій послідовності відповідно до пріоритетів виконуються операції?

3. ФОРМАТНИЙ ВВІД/ВИВІД ПРИ РОБОТІ З КОНСОЛЛЮ

Навчальною метою розділу є ознайомлення читача з функціями форматного вводу/виводу.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– вводити значення з консолі у програму;
– виводити константні та форматовані рядки на консоль;
– застосовувати специфікатори для форматування рядків;
– задавати точність та поле виводу для значень, що виводяться на консоль.

3.1. Використання функцій форматного вводу/виводу

Щонайменш половина програм, що розробляються, орієнтовано на користувача. Це означає, що в них реалізовано дружній інтерфейс для спілкування з людиною. Призначення такого інтерфейсу – дізнатися від користувача за допомогою підказок та запитів про всі особливості задачі, що повинна бути вирішена. Серед таких програм консолі є найбільш простими.

Ввід та вивід даних за допомогою консолі в С++ може бути організовано декількома шляхами. Один з них – використання форматних функцій printf() та scanf(). Застосування цих функцій стає доступним після підключення до програми заголовного файлу stdio.h.

3.1.1. Функція форматного виводу printf()

Форматна функція printf() призначена для виводу форматованого рядка на консоль. Форматування рядка виконується безпосередньо у самій функції. Нижче наведено формат виклику printf():

printf(<*рядок, що форматується* >,<*список значень* >)

Список значень є опціональною можливістю функції, причому за його наявності кількість змінних невід'ємно залежить від використаних у рядку, що формується, специфікаторів. За відсутності списку значень printf() виводить на консоль статичний рядок.

Наприклад, наступний виклик функції:

```
printf("Some text");                    //вивід константного рядка
```

приведе до появлення на консолі рядка, що вказано у подвійних лапках.

Але статичні рядки це лише незмінний текст, програми ж виконуються для отримання значень, що є результатами обчислень чи логічних висновків. Для того, щоб вивести будь-які динамічні дані у функції printf(), застосовуються специфікатори перетворення, що наведені у табл. 3.1.

Таблиця 3.1

Специфікатори функції форматного виводу printf()

Специфікатор	**Значення**
%c	символ
%d	десяткове ціле число зі знаком
%i	десяткове ціле число зі знаком
%e	науковий формат (буква нижнього регістру e)
%E	науковий формат (буква верхнього регістру e)
%f	десяткове число з плаваючою точкою
%g	в залежності від того, який формат коротше, застосовується або %e, або %f
%G	в залежності від того, який формат коротше, застосовується або %e, або %f
%o	вісімкове число без знаку
%s	рядок символів
%u	десяткове число без знаку
%x	шістнадцяткове число без знаку (букви нижнього регістру)
%X	шістнадцяткове число без знаку (букви верхнього регістру)
%p	покажчик
%n	покажчик на цілу змінну. специфікатор викликає присвоювання цій цілій змінній кількості символів, виведених перед нею
%%	знак %

Розглянемо на прикладі як використовується вивід значень змінних за допомогою специфікаторів та функції printf()

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

void main()                           //оголошення та визначення головної функції
{
        int i=10;                     //оголошення та визначення цілої змінної
        float f=7.12; //оголошення та визначення змінної з плаваючою точкою
        char c='F';                //оголошення та визначення символьної змінної
        printf("i=%i\r\n",i);         //вивід значення на консоль
        printf("f=%f\r\n",f);         //вивід значення на консоль
        printf("c=%c\r\n",c);         //вивід значення на консоль
}
```

Для використання функції printf() через директиву #**include** виконується підключення заголовного файлу stdio.h. У головній функції оголошуються змінні трьох типів: цілого i, з плаваючою точкою f та символьного c. Значення змінним присвоюються при оголошенні. Три останніх рядки є викликами

функції printf(), у яких відповідні значення форматуються у рядки і виводяться на консоль.

Результатом роботи програми буде:

```
i=10
f=7.12
c= F
```

Кожний з рядків, що форматуються, у прикладі закінчується на сполучення команд "\r\n". Це сполучення дозволяє компілятору зрозуміти, коли на консолі повинен з'явитися перехід на новий рядок. Таких команд форматування багато. Найбільш поширені з них:

\r – повернення каретки (походить від роботи з друкарською машинкою);

\n – перехід на новий рядок;

\t – табуляція;

\\ – знак зворотного слеша.

Для більшості компіляторів для виконання переходу на новий рядок достатньо однієї команди "\n", але для однозначного виконання краще задіяти сукупність команд "\r\n", як було наведено у прикладі.

Гнучкість застосування специфікаторів перетворення може бути збільшено за допомогою модифікаторів формату: мінімальної ширини поля та точності.

Модифікатори формату вказуються через крапку між символом "%" та позначкою відповідності специфікатора. Крапка є частиною модифікатора точності і при наявності лише модифікатора мінімальної ширини поля – відсутня.

Модифікатор мінімальної ширини поля задає яка найменша кількість розрядів буде використана при виводі числа на консоль, а модифікатор точності вказує точність, з якою буде надано число. Модифікатор мінімальної ширини поля може мати від'ємне значення, щоб виконувати вирівнювання по лівій границі. Розряди, що не задіяні, залишаються порожніми.

Наступний приклад демонструє особливості застосування модифікаторів формату для різних специфікаторів.

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

void main()                           //оголошення та визначення головної функції
{
      printf("%.2f \r\n", 15.17432);  //вивід значення з крапкою, що плаває
      printf("%6f\r\n", 15.17432);    //вивід значення з крапкою, що плаває
      printf("%6.2f\r\n",15.17432);   //вивід значення з крапкою, що плаває
      printf("%-6.2f\r\n",15.17432);  //вивід значення з крапкою, що плаває
      printf("%.3i\r\n", 2);          //вивід цілого значення
      printf("%5i\r\n", 2);           //вивід цілого значення
      printf("%5.3i\r\n", 2);         //вивід цілого значення
```

```
	printf("%-5.3i\r\n", 2);		//вивід цілого значення
	printf("%15s\r\n", "Some text");	//вивід рядка
	printf("%-15s\r\n", "Some text");	//вивід рядка
	printf("%.15s\r\n", "Some text for console test"); //вивід рядку
}
```

Для роботи з форматними функціям підключено stdio.h. *У головній функції виконується послідовний виклик* printf(). *Чотири виклики опрацьовують значення з плаваючою точкою; наступні чотири виклики форматують значення цілого типу; останні три виклики виводять статичний рядок.*

Результатом виконання коду стануть наступні рядки:

1	5	.	1	7											точність 2 розряди
1	5	.	1	7	4	3	2								мінімальна кількість розрядів 6
	1	5	.	1	7										мінімальна кількість 6, точність 2 розряди
1	5	.	1	7											попереднє, вирівняно по лівій межі
0	0	2													точність 3 розряди
				2											мінімальна кількість розрядів 5
		0	0	2											мінімальна кількість 5, точність 3 розряди
0	0	2													попереднє, вирівняно по лівій межі
						S	o	m	e		t	e	x	t	мінімальна кількість розрядів 15
S	o	m	e		t	e	x	t							попереднє, вирівняно по лівій межі
S	o	m	e		t	e	x	t		f	o	r		c	точність 15 розрядів

Як можна зрозуміти з прикладу, при нестачі у числі значущих розрядів воно доповнюється нулями, причому для цілих значень це виконується зліва, а для значень з крапкою, що плаває, – справа. На статичний текст модифікатори формату також впливають, причому при невідповідності кількості символів модифікатора точності виводиться тільки частина, починаючи зліва.

3.1.2. Функція форматного вводу scanf()

Форматна функція scanf() призначена для вводу даних з консолі. Формат її виклику наведено нижче:

scanf(<*рядок, що форматується* >,<*список адрес змінних* >);

Список адрес змінних, як і для printf() список значень, несе опціональний характер, але застосовується майже завжди, бо відповідає основному призначенню.

Статичний текст у рядку, що форматується, теж не є потрібним. Специфікатори перетворення, що можуть бути застосовані для форматування рядка, наведено у табл. 3.2.

Таблиця 3.2

Специфікатори функції форматного вводу scanf()

Специфікатор	Значення
%c	символ
%d	десяткове ціле число зі знаком
%i	десяткове ціле число зі знаком, вісімкове або шістнадцяткове число
%e	десяткове число з крапкою, що плаває
%f	десяткове число з крапкою, що плаває
%g	десяткове число з крапкою, що плаває
%o	вісімкове число
%s	рядок символів
%u	десяткове число без знаку
%x	шістнадцяткове число без знаку (букви нижнього регістру)
%p	покажчик
%n	покажчик на цілу змінну. Специфікатор викликає присвоювання цій цілій змінній кількості символів, виведених перед нею
%[]	набір символів, що скануються
%%	знак %

Частина перелічених специфікаторів відповідають printf().

Розглянемо особливості використання функції scanf() на прикладі програми, що отримує дані від користувача і відображає їх на консолі.

```
#include <stdio.h>                  //підключення бібліотеки вводу/виводу

void main()                         //оголошення та визначення головної функції
{
    int i;                          //оголошення цілої змінної
    float f;                        //оголошення змінної з плаваючою точкою
    char c;                         //оголошення символьної змінної
    printf("Enter int value: ");    //вивід запиту до користувача
    scanf("%i", &i);                //отримання значення від користувача
    printf("int value is: %i \r\n", i); //вивід значення на консоль
    printf("Enter float value: ");  //вивід запиту до користувача
    scanf("%f", &f);                //отримання значення від користувача
    printf("float value is: %f \r\n", f); //вивід значення на консоль
    printf("Enter char value: ");   //вивід запиту до користувача
    scanf("%s", &c);                //отримання значення від користувача
    printf("char value is: %c \r\n", c); //вивід значення на консоль
}
```

Для роботи з форматними функціями підключено stdio.h. *У головній функції оголошені змінні:* i *– цілого типу,* f *– типу з точкою, що плаває,* c *– символьного типу. Програма виконує почергове запитування значень для вказаних змінних від користувача і після їх вводу виводить отримані дані на консоль.*

При вводі послідовності 3; 7.36; A буде отримано наступний результат:

```
Enter int value: 3
int value is: 3
Enter float value: 7.36
float value is: 7.36
Enter char value: A
char value is: A
```

Таким чином, використання специфікаторів перетворення функції scanf() відповідає вже розглянутому для printf().

Для функції форматного вводу також є модифікатори, але їх застосування скоріше ускладнює використання функції. Також scanf() краще використовувати для отримання протягом одного разу лише єдиного значення. Ввести сукупність даних можливо, але для цього треба дотримуватися правил, що обтяжують програмний код.

На завершення треба приділити увагу тому як зазначається у функції scanf() змінна, до котрої надходить значення. Перед її ім'ям дописується операція "&", що не є ні логічною, ні порозрядною, бо використовується унарно. Ця операція називається отриманням адреси. Більш детальніше її буде розглянуто у темі "Змінні-покажчики". Якщо при написанні програми опустити операцію отримання адреси, то це не буде сприйнято компілятором, як помилка, але значення, що буде введено, до змінної не потрапить, тому застосування "&" є обов'язковим.

Висновки

У даному розділи було розглянуто наступні основні питання:

– функція форматного виводу printf();

– функція форматного вводу scanf().

Контрольні питання

1) У якому заголовному файлі знаходяться описи форматних функцій вводу/виводу?

2) Яка функція застосовується для виводу на консоль форматованого рядка?

3) Що називають форматованим рядком?

4) Що таке специфікатор форматування?

5) Який специфікатор дозволяє вивести значення цілочисельного типу подвійної довжини?

6) Який специфікатор дозволяє вивести значення у науковому форматі?

7) Для чого застосовується специфікатор %n?

8) Як вивести ціле число у вісімковій та шістнадцятковій формах?

9) Що таке команда форматування?

10) Які є модифікатори для функції форматного виводу?

11) Як вивести число у поле з п'яти розрядів, вирівнявши його по лівому краю?

12) Як доповнити ціле число нулями до відповідної кількості розрядів?

13) Як виконати урізання тексту до заданої кількості символів?

14) За допомогою чого можна задати точність дійсних значень?

15) Яке призначення функції scanf()?

16) Чому на відміну від printf() у scanf() другий параметр найчастіше не є опціональним?

17) Чому scanf() найчастіше застосовується для вводу лише одного значення?

18) Як за допомогою функції форматного вводу ввести символ?

19) Чому для другого параметру scanf() застосовується оператор "&"?

20) Як за допомогою printf() вивести символ "%"?

4. РОЗГАЛУЖЕННЯ

Навчальною метою розділу є ознайомлення читача з розгалуженнями програмного коду на основі операторів та операції умови.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– створювати за допомогою складеного оператору умови дві гілки реалізації програмного коду;
– реалізовувати на основі оператору **if** сходинкову програмну структуру;
– застосовувати оператор вибору для перевірки значень цілих змінних;
– виконувати безумовний вихід з оператору switch;
– використовувати умовну операцію.

4.1. Оператори умови

Досить рідко в задачу програми входить виконання лише ряду послідовних дій. Це пов'язано з тим, що алгоритмізація вимагає найбільшої подібності з ситуаціями реального життя, в якому все прийнято ототожнювати з подією. Кожен день перед нами постає вибір, причому прийняття рішення рідко буває спонтанним, частіше воно залежить від цілої послідовності факторів. Таким чином, сукупність факторів призводить до своєрідного розгалуження можливого розвитку подій. Для людини події дня минають послідовно, але, обертаючись назад, ми часто розмірковуємо, а що могло б відбутись, якщо б прийняте рішення можна було б змінити?

Іноді життя дає можливість роз'яснити це, а інколи збіг обставин призводить до того, що подія, яка відбулась один раз, більше ніколи не повториться. Та все ж таки, більшість задач, що виконуються людиною, з невеликою відмінністю повторюються, хоч і за різним степенем регулярності. Так, кожен день, усі дії людини залежать від прийнятого розкладу: приймання їжі, робота або навчання, тривалість сну. Щось повторюється щоденно, щось щотижнево, а щось і ще з більшою періодичністю.

Усе у нашому житті залежить від збігу обставин, однак у загальному випадку можна скласти алгоритм, відповідно до якого існує високий степінь ймовірності описати події дня для конкретної людини.

Таким чином, алгоритмізація зумовлює наявність як розгалужень, так і повторень. У мові С++ для формування множини варіантів виконання програми, які залежать від зробленого вибору, використовуються умовні оператори. Вони поділяються на дві групи: оператори розгалужень та оператори циклів.

Для реалізації програмних гілок використовуються оператори **if-else**, **switch** та умовна операція "**?:**".

4.1.1. Складений оператор умови if-else

if-else прийнято називати складеним оператором умови. Префікс складений говорить про те, що оператор умови включає одразу два службових слова: **if** та **else.** Даний оператор дає змогу організувати в програмі дві гілки,

вибір однієї із яких на протязі виконання виконується за рахунок визначеної умови.

Складений оператор умови відображається у схемах алгоритмів за допомогою блоку рішення (рис. 4.1).

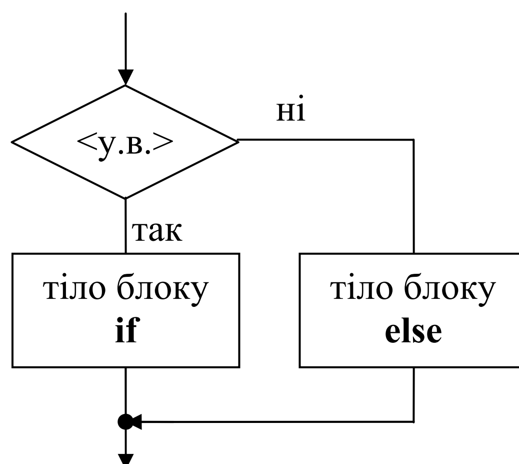

Рис. 4.1. Відображення оператора умови у схемах алгоритму

Вибір гілки, за якою буде минати подальше виконання програми, виконується за рахунок перевірки умови на істинність. В програмі оператор **if-else** має наступний формат:

if (<*умовний вираз*>)
<*тіло **if***>
else
<*тіло **else***>

Оператори мов С та С++ можна умовно розділити на дві групи: прості та складні. До простих відносяться **break**, **continue**, **goto**, а до складних: **if-else**, **switch**, **while**, **do-while**, **for**. Прості оператори є сформованими командами, які несуть у собі однозначну вказівку. Використовуючи такі оператори, програміст не може вплинути на процес їх виконання.

На відміну від простих, складні оператори без втручання програміста зовсім не виступають командами. Такі оператори завжди включають один або декілька параметрів, взятих у круглі дужки, які повинні в обов'язковому випадку бути визначеними програмістом під час формування програми. Параметри використовуються для того, щоб програміст міг явно задавати умови виконання інструкцій тіла оператора. Для оператору **if-else** параметром є умовний вираз.

Умовний – це будь-який вираз, результат підрахунку якого дорівнює одному із двох значень: 0 або 1. Розуміється у тому сенсі, що в якості такого виразу можливо використовувати змінні типу **bool** або вирази, які включають операції відношення та логічні операції. Та все ж, це правило не суворе. Компілятор не видасть помилку, незважаючи на тип даних змінної або операцій, з якими вираз використовує програміст. Тобто помилкою не буде

поставити в круглі дужки змінну N дійсного типу даних зі значенням 359.17 або вираз z + 21/3. Така вільність написання ніяк не може призвести до припинення виконання програми, оскільки для компілятора, коли під час розбору програми зустрічається оператор **if-else**, має значення величина та тип значення. Він розуміє лише два: 0 та 1. Таким чином, усе, що не відповідає цим значенням, компілятором автоматично приводиться до 0. Тобто можна сказати, що усі значення, окрім нуля, для умовного виразу оператора **if-else** приймаються за одиницю.

Даний підхід має лише один недолік, – якщо в програмі умовний вираз містить логічну помилку, тоді, через ігнорування компілятором, її виявлення стає досить складною задачею.

Складений оператор умови **if-else** на відміну від інших складних операторів має два тіла. При істинності умовного виразу (значення 1) виконується тіло **if**, а при хибності (значення 0) – тіло **else.** Тілом оператора прийнято називати інструкцію або програмний блок, який включає групу інструкцій. Таким чином, число інструкцій, які необхідно виконати у відповідній гілці програми, визначає наявність або відсутність фігурних дужок в тілах оператора **if-else**. Зустрічаючи після службового слова складного оператора відкриваючу фігурну дужку, компілятор розуміє, що всі інструкції до відповідної закриваючої дужки відносяться до тіла цього оператора.

Складена частина **if** слугує для того, щоб сформувати програмну гілку, яка буде містити код, що виконується при одиничному значенні умови, а **else** – при нульовому. Таким чином, гілка **else** буде виконуватись при будь-якому результаті, який не відповідає поставленій умові. Дуже часто оператор **if-else** використовується для розділення програмної обробки різних діапазонів чисел.

Розглянемо використання оператора **if-else** на прикладі програми, яка дозволяє вивести на консоль значення модуля для введених користувачем цілих чисел:

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

void main ()                          //оголошення та визначення головної функції
{
        int n;                        //оголошення змінної для отримання значення
        printf("Enter the value: ");  //вивід запиту до користувача
        scanf("%i", &n);              //отримання значення від користувача
        if(n < 0)                     //якщо значення від'ємне
                printf("Module of %i is %i", n, -n);  //вивід значення модулю
        else                          //якщо значення позитивне
                printf( "Module of %i is %i", n, n);  //вивід значення модулю
}
```

У наведеній програмі оператор **if-else** *використано для того, щоб розділити діапазон цілих значень, які вводяться користувачем, на додатні та від'ємні. Якщо було введено від'ємне значення, тоді умовний вираз виконається і*

користувачу буде відображено результат операції – унарний мінус для змінної n*. В іншому випадку значення* n *буде виведено без виконання над ним будь-яких дій.*

Тіла оператора **if-else** включають по одній інструкції, тому фігурні дужки відсутні. Важливо розуміти, що використання фігурних дужок для виділення в тіло оператора однієї інструкції, не є помилкою. Частіше усього написання без дужок використовується просто для зменшення довжини програмного коду. Однак нерідко, така форма оператора **if-else** при включенні у якості інструкції іншого складеного оператора умови може призвести до логічних помилок, які компілятор не виявляє. Тому при запам'ятовуванні особливостей роботи оператора **if-else** бажано використовувати фігурні дужки незалежно від числа інструкцій, які включені у тіло. Крім цього, відсутність дужок при двох і більше інструкціях однозначно буде являтись синтаксичною або логічною помилкою.

Складений оператор умови **if-else** на відміну від усіх інших складних операторів може мати як повну, так і скорочену форму. В скороченій формі гілка, що виділена службовим словом **else,** відсутня:

```
if(<умовний вираз>)
    <тіло if>
```

Така форма демонструє, що у випадку невиконання поставленої умови, ніяких дій робити непотрібно.

Наприклад, якщо від програми вимагається визначити чи є введене користувачем число кратне трьом, тоді код матиме наступний вигляд:

```
#include <stdio.h>                  //підключення бібліотеки вводу/виводу

void main ()                        //оголошення та визначення головної функції
{
    int n;                          //оголошення змінної для отримання значення
    printf("Enter the value:");     //вивід запиту до користувача
    scanf("%i", &n);                //отримання значення від користувача
    if(n%3==0)                      //якщо значення є кратним трьом
        printf("Value is divided by 3");//вивід константного рядка
}
```

В даній програмі для виконання перевірки кратності трьом використовується операція ділення за модулем (знаходження залишку від ділення) для значення змінної n і подальша перевірка відповідності отриманого результату до нуля. Якщо умову виконано, на консоль буде виведено повідомлення, яке інформує користувача, що він ввів значення, яке є кратним трьом. В іншому випадку, програма просто завершиться.

Таким чином, складений оператор умови може мати як повну форму **if-else**, так і скорочену – **if.** Обидві форми в програмах використовуються з однаковою регулярністю. Найчастіше скорочена форма **if-else** використовується у сукупності з оператором циклів.

Як бачимо з прикладу, правило про фігурні дужки для тіла оператора **if-else** також відноситься і до його скороченої форми.

Складений оператор умови також інколи називають сходинковим, що характеризує вид програми з формуванням при вкладці одного оператора **if-else** в другий. Такий підхід використовується для послідовної перевірки декількох умов. Сходинковість такого розгалуження програми також добре помітна і на схемах алгоритму.

Розглянемо сходинковий підхід до використання операторів **if-else** на прикладі програми, яка дозволяє перевірити чи є введене користувачем число парним, від'ємним та кратним п'яти, а також виконати його піднесення у другий степінь з подальшим виводом результату на консоль:

```
#include <stdio.h>                  //підключення бібліотеки вводу/виводу

void main()                         //оголошення та визначення головної функції
{
    int n;                          //оголошення змінної для отримання значення
    printf("Enter the value:");     //вивід запиту до користувача
    scanf("%i", &n);                //отримання значення від користувача
    if(n < 0)                       //якщо значення є від'ємним
        if(n%2 == 0)                //якщо значення є кратним 2
            if(n%5 == 0)            //якщо значення є кратним 5
            {
                n*=n;               //розрахунок квадрату значення
                printf("Square of n is %i", n); //вивід результату
            }
            else                    //якщо значення не є кратним 5
//вивід константного рядка
                printf("Is not divided by 5!");
        else                        //якщо значення не є кратним 2
            printf("Is not divided by 2!"); //вивід константного рядка
    else                            //якщо значення не є від'ємним
        printf("Positive!");                    //вивід константного рядка
}
```

Як видно з програми, послідовна перевірка умов і використання форматування приводять до формування із операторів **if-else** своєрідної сходинкової структури. Подібна структура спостерігалась і на схемі алгоритму.

Однак у формуванні такої структури досить часто просто немає необхідності. Перевірені умови при вкладці у кожен зовнішній оператор **if-else** тільки однієї конструкції, в якості якої також виступає складений оператор

умови, може бути замінено одним оператором, умовний вираз котрого буде характеризувати всі необхідні перевірки.

Однак при цьому складена частина **else** також буде лише одна, що не дозволить організувати множинне розгалуження для варіантів розвитку програми при порушенні однієї із умов. В наведеному прикладі підхід реалізації шляхом заміни усіх операторів **if-else** на:

```
if(n<0 && n%10 == 0)          //якщо значення є від'ємним  та кратним 2 та 5
{
        n*=n;                           //розрахунок квадрату значення
        printf("Square of n is %i ", n); //вивід результату
}
else                            //якщо значення не є від'ємним та не кратним 2 та 5
//вивід константного рядка
        printf("Value is positive or not divided by 2 or 5!");
```

Даний код дозволяє позбавитись трьох умов шляхом заміни їх одним виразом, який перевіряє, щоб число було від'ємним і в той самий час ділилося на 10. *Але введене число тепер стало неможливо контролювати на помилки, оскільки, якщо раніше кожна невідповідність умові призводила до виводу на консоль конкретного зауваження, то зараз на будь-яку хибну ситуацію користувачу виведеться лише загальна фраза*.

Поєднання декількох сходинково-структурованих операторів **if-else** стає неможливим, якщо в одному з їх тіл присутні дві чи більше інструкцій.

Наприклад, якщо для вище наведеної програми при послідовній перевірці характеристик введеного числа необхідно було б виконувати над ними корегуючи дії:

```
if(n < 0)                               //якщо значення від'ємне
{
        n*=-1;                          //зробити значення позитивним
        if(n%2 == 0)                    //якщо значення є кратним 2
        {
                n*=2;                   //помножити значення на 2
                if(n%5 == 0)            //якщо значення є кратним 5
                {
                        n*=n;           //розрахунок квадрату значення
                        printf("Square of modified n is %i ", n);       //вивід результата
                }
                else                    //якщо значення не є кратним 5
                        printf("Is not divided by 5!"); //вивід константного рядку
        }
        else                            //якщо значення не є кратним 2
                printf("Is not divided by 2!");         //вивід константного рядка
```

```
}
else                                          //якщо значення не є від'ємним
        printf("Positive!");                  //вивід константного рядка
```

Під час використання сходинкової структури оператора **if-else** в сукупності зі скороченою формою одного або декількох вкладених складених операторів умови може приводити до логічних помилок, які залежать від фігурних дужок тіл.

Наприклад, розглянемо програму, яка визначає чи відноситься до букв символ, що знаходиться у таблиці ASCII кодів під введеним користувачем номером:

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

void main()                           //оголошення та визначення головної функції
{
        int n;                  //оголошення змінної для отримання значення
        printf("Enter the code of ASCII character:"); //вивід константного рядка
        scanf("%i", &n);              //отримання значення від користувача
        if(n>= 0 && n <= 255)
//якщо значення це буква
                if((n >= 'A' && n <= 'z') || ( n >= 'a' && n <= 'z'))
                        printf("The character is a letter"); //вивід константного рядка
        else                          //якщо значення не відповідає діапазону
                printf("The code is not ASCII!");       //вивід константного рядка
}
```

Дана програма містить логічну помилку, яка основана на відсутності дужок для тіл складеної частини **if**. Вона полягає у тому, що хоч форматування і було зроблено коректно, але для компілятора складена частина **else** завжди відноситься до попереднього службового слова **if,** яке знаходиться на тому самому рівні вкладеності. Таким чином, складена частина **else** в наведеній програмі буде віднесена компілятором не до зовнішнього складеного оператора умови, а до вкладеного, і вивід повідомлення про невідповідність діапазону **ASCII** введеного номеру буде з'являтись навіть при кодах **ASCII** символів, які не є буквами. У великих програмах подібні помилки помітити досить важко, тому при формуванні сходинкової структури на основі **if-else** при вкладеності у тіло **if** бажано не нехтувати дужками навіть тоді, коли в них немає явної потреби.

Для наведеного раніше прикладу правильна сходинкова структура оператора **if-else** має мати такий вигляд:

```
if(n >= 0 && n <= 255)          //якщо значення у діапазоні від 0 до 255
{
//якщо значення це буква

        if((n >=  'A' && n <= 'z') || ( n >= 'a' && n <= 'z'))
                printf ("The character is a letter");     //вивід константного рядка
}
else                                    //якщо значення не відповідає діапазону
        printf ("The code is not ASCII!");            //вивід константного рядка
```

4.1.2. Конструкція else if

Базуючись на сходинковій структурі складеного оператора умови, можна організувати послідовну перевірку декількох умов.

Розглянемо програму, яка виконує функції найпростішого калькулятора для цілих чисел:

```
#include <stdio.h>                      //підключення бібліотеки вводу/виводу

void main()                             //оголошення та визначення головної функції
{
        int a, b;                       //оголошення змінних для отримання значень
        char sign;                      //оголошення змінної для отримання значення
        printf ("Enter the a value: ");    //вивід запиту до користувача
        scanf ("%i ", &a );             //отримання значення від користувача
        printf ("Enter the sign of operation: "); //вивід запиту до користувача
        scanf ("%s ", &sign);           //отримання значення від користувача
        printf ("Enter the b value: ");    //вивід запиту до користувача
        scanf ("%i ", &b );             //отримання значення від користувача
        if(sign == '+')                 //якщо введено "+"
                printf ("a + b = %i", a + b);      //вивести результат суми
        else if(sign == '-')            //якщо введено "-"
                printf ("a – b = %i", a – b);      //вивести результат віднімання
        else if(sign == ' * ')          //якщо введено "*"
                printf ("a * b = %i", a * b);      //вивести добуток
        else if(sign == ' / ')          //якщо введено "/"
        {
                if(b != 0)              //якщо дільник не має значення 0
                        printf ("a / b = %f ", a * 0.1/b); //вивести результат ділення
                else                    //якщо дільник має значення 0
                        printf ("Error! Dividing by 0!"); //вивід константного рядка
        }
        else                            //якщо введено інший символ
                printf ("Sign of operation is not valid!"); //вивід константного рядка
}
```

Користувачу пропонується виконати послідовний ввід значень для змінної a*, знаку операції та значення для змінної* b*. Потім відбувається послідовне порівняння значення змінної sign зі знаками операції, що підтримуються. Для операції ділення також перевіряється нерівність дільника нулю. Результат операції виводиться на консоль.*

У наведеній програмі використовується так названа конструкція else if. Ця конструкція є поєднанням службових слів під час реалізації сходинкової структури операторів **if-else.** В даному випадку вкладення здійснюються не у тіло **if**, а у тіло **else.** При такій організації прийнято не ставити фігурні дужки для явного поміщення у тіло **else** вкладеного оператора **if-else** і виконувати запис слів **else** та **if**, які формують сходинку, в одному рядку. Конструкція **else if** дозволяє виконати перебір як окремих значень змінної, що контролюється, так і діапазони, в які вона може входити. Дана конструкція, що являє собою окремий випадок сходинкової структури, відрізняється від неї значно меншою вірогідністю виникнення логічних помилок, що досягається за рахунок використання **else if** для конкретних задач і повної відповідності правилом використання оператора **if-else.**

Розглянемо програму, яка виконує порівняння введених користувачем значень:

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

void main()                           //оголошення та визначення головної функції
{
        int a, b;                     //оголошення змінних для отримання значень
        printf("Enter the a value: ");    //вивід запиту до користувача
        scanf("%i", &a);              //отримання значення від користувача
        printf("Enter the b value: ");    //вивід запиту до користувача
        scanf("%i", &b);              //отримання значення від користувача
        if(a > b)                     //якщо а більше за b
                printf("%i > %i", a, b);  //вивести результат
        else if(a < b)                //якщо а менше за b
                printf("%i < %i", a, b);  //вивести результат
        else                          //якщо а дорівнює b
                printf("%i = %i", a, b);  //вивести результат
}
```

Після вводу користувачем значень a *і* b *відбувається послідовна перевірка можливих відносин між ними. Коли відношення між* a *і* b *відповідають умовній гілці, виконується вивід на консоль відповідного повідомлення.*

Серед операторів умови оператор **if-else** використовується найчастіше. Це забезпечується його простотою та універсальністю сформованих ним програмних розгалужень.

4.1.3. Оператор вибору switch

Ще один оператор умови **switch** хоч і використовується значно рідше ніж **if-else**, але для деяких задач програмування просто незамінний. Його прийнято називати оператором вибору.

Для його відображення на схемах алгоритмів використовується блок вирішення окремої реалізації:

На схемах розгалуження на основі **switch** також нагадує сходинкову структуру (рис. 4.2), однак у програмі такої не спостерігається.

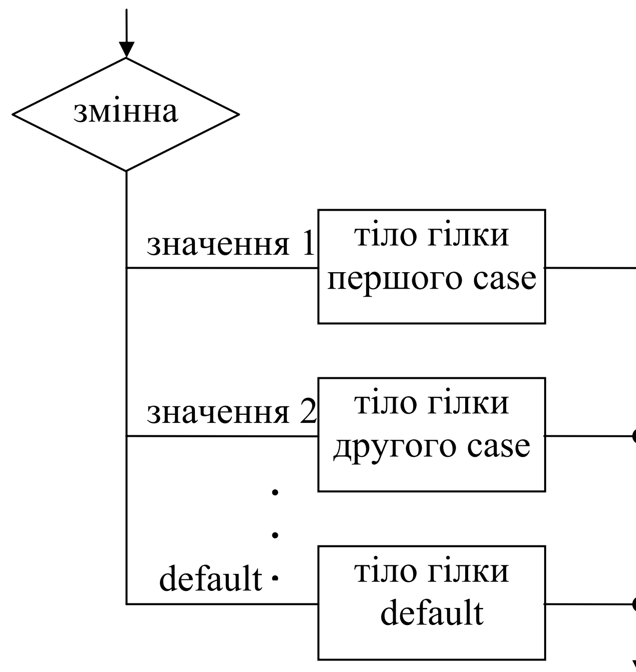

Рис. 4.2. Відображення оператору вибору у схемах алгоритму

Оператор **switch** дозволяє виконати перевірку відповідності змінної цілого типу одному з перебраних значень. Збіг слугує причиною для виконання визначеного гілкою програмного коду. Формат оператора **switch**:

```
switch (<змінна цілочисельного типу даних>)
{
	case <значення 1>:
		<група операторів>
		break;
	case <значення 2>:
		<група операторів>
		break;
	default:
		<група операторів>
		break;
}
```

Параметром оператора **switch** слугує змінна цілого типу. Будь-який інший тип даних змінної для оператора викличе помилку. Для **switch** правило про параметр суворе. Параметром можуть бути тільки змінні цілих типів даних (наприклад, **char** та **int**).

Кожен допоміжний оператор **case** виконує функцію формування однієї з програмних ймовірних гілок. Умовний вираз у явній формі для оператора **switch** відсутній. Він формується динамічно у той час, коли при виконанні програми в операторі **switch** зустрічається черговий **case**. Такий умовний вираз засновано лише на одній операції порівняння – "= =". Таким чином, здійснюється перевірка на збіг значення змінної, що перевіряється, із вказаним за допоміжним оператором **case** константним значенням. Якщо значення збігаються, то виконується програмний код, що розташовано між поточним та наступним **case**. В іншому випадку перевіряється значення чергового допоміжного оператора **case**.

Кількість ймовірних розгалужень, що може бути створено за допомогою оператора вибору, обмежено лише особливостями компілятора. Створення гілок з однаковими константними значеннями не є помилкою, але не має логічного сенсу.

Для того, щоб зменшити, при можливості, час використання оператора **switch**, наприкінці кожної з гілок приписується оператор безумовного завершення **break**, що здійснює вихід з тіла оператору вибору. Таким чином, якщо при переборі умов було реалізовано якусь з гілок, то на цьому виконання оператору **switch** закінчується. Використання оператору **break** не є обов'язковим, якщо у **switch** відсутня гілка **default**.

Допоміжний оператор **default** також, як і оператор **case**, формує ймовірну гілку **switch**, але не має зв'язаного константного значення. **default** реалізує останню гілку і умова її виконання передбачає всі можливі значення. При наявності операторів **break** наприкінці всіх гілок **case**, гілка **default** приймає сутність – у випадку якщо поточне значення не збігається із жодним з перелічених константних значень. Наявність гілки **default** не є обов'язковою, але у цьому випадку може виникати ситуація, коли не буде виконано жодної з гілок оператора вибору. Допоміжний оператор **default** впроваджує для **switch** своєрідну універсальність.

Гілка, що формується оператором **default**, не потребує наприкінці застосування оператору **break**, бо є останньою, і після неї вихід з **switch** буде виконано у будь-якому випадку. Але хоч **break** у цій гілці і не має сенсу, його використання не буде сприйнято, як помилку.

Розглянемо приклад програми, в якій оператор вибору застосовано для реалізації функцій простого калькулятора.

```
#include <stdio.h>              //підключення бібліотеки вводу/виводу

void main()                     //оголошення та визначення головної функції
{
        float a,b;              //оголошення змінних для вводу значень
```

```
    char sign;                    //оголошення символьної змінної
    printf("Calc a x b (x values: *,/,+,-)\r\n"); //вивід константного рядка
    printf("a value is:");        //вивід константного рядка
    scanf("%f", &a);              //отримання значення змінної a
    printf("x value is:");        //вивід константного рядка
    scanf("%s", &sign);           //отримання значення змінної sign
    printf("b value is:");        //вивід константного рядка
    scanf("%f", &b);              //отримання значення змінної b
    printf ("%f%c%f= ", a, sign, b);      //вивід отриманого виразу
    switch(sign)                          //перевірка значення змінної sign
    {
        case '+':                         //якщо символ '+'
            printf("%f", a+b);            //вивід результату суми
            break;                        //вихід з оператора
        case '-':                         //якщо символ '-'
            printf("%f", a-b);            //вивід результату різниці
            break;                        //вихід з оператора
        case '*':                         //якщо символ '*'
            printf("%f", a*b);            //вивід добутку
            break;                        //вихід з оператора
        case '/':                         //якщо символ '/'
            if(b!=0)                      //якщо значення b не 0
                printf("%f", a/b);        //вивід результату ділення
            else                          //якщо значення b – 0
                printf("_ (Division by 0)"); //вивід константного рядка
            break;                        //вихід з оператора
        default:                          //якщо інший символ
            printf("_ (Not valid x value)"); //вивід константного рядка
    }
}
```

Програма повідомляє користувача про операцію, що буде виконано, а потім по черзі приймає значення для змінних a, sign *та* b. *Формула, що була створена таким чином, виводиться на консоль, і починається пошук гілки, значення якої відповідає обраному користувачем. Коли гілку знайдено, то розраховується результат. Перевірка на "*0*" та гілка* **default** *виконують функцію захисту від хибного вводу.*

4.1.4. Оголошення змінних у гілках case

Формування гілок за допомогою операторів **case** і **default** виконується у **switch** без використання програмних блоків. Це треба враховувати, якщо буде необхідність в оголошенні у одній з гілок змінної. Компілятор неспроможний прийняти оголошення змінної, що відповідає рівню тіла **switch**, тому для того, щоб запобігти помилок необхідно весь програмний код поточного **case** помістити у неіменований програмний блок.

Наприклад:

```
switch(a)
{
	case 1:
	{
		int z;
		.
		.
		.
		break;
	}
	.
	.
	.
}
```

4.2. Умовна операція "?:"

Операція "?:" також, як і оператори **if-else** та **switch**, дозволяє виконати розгалуження програмного коду. Але на відміну від операторів, умовна операція застосовується у виразах. Реалізацію умовної операції у програмі відображує наведений нижче формат:

<*умовний вираз*> ? <*вираз А*> : <*вираз Б*>

Якщо результатом умовного виразу є 1, то виконується вираз А, у іншому випадку – вираз Б.

Принцип дії "?:" відповідає приватному випадку оператора **if-else**, коли в тілах **if** та **else** реалізується по одному виразу.

Таким чином, умовна операція може бути використана у разі необхідності скорочення програмного коду чи замість оператора **if-else**.

Розглянемо приклад програми, що знаходить значення модуля для введеного числа:

```
#include <stdio.h>                //підключення бібліотеки вводу/виводу

void main()                       //оголошення та визначення головної функції
{
	int x;                        //оголошення змінної для вводу значення
	printf("Enter the X value:");     //вивід запиту до користувача
	scanf("%i", &x);              //отримання значення від користувача
	printf("|%i|=%i \r\n", x, (x<0) ? (-x) : x);      //вивід результату на консоль
}
```

Програма запитує у користувача значення для змінної x, обчислює його модуль і виводить результат на консоль. Необхідності у круглих скобках при записі виразів умовної операції немає, але їх присутність часто запобігає логічним помилкам, та підвищує розуміння програмного коду.

Висновки

У даному розділи було розглянуто наступні основні питання:

– складений оператор умови if-else;

– оператор вибору switch;

– умовна операція "?:".

Контрольні питання

1) Які оператори та операції у мові C++ виконують розгалуження виконання коду?

2) Чому складений оператор умови називають сходинковим?

3) У чому полягає зміст застосування оператору **if-else**?

4) Чому **if-else** називають складеним?

5) У якому випадку тіла **if-else** можна не брати у фігурні дужки?

6) У яких випадках застосування фігурних дужок для складеного оператору умови є необхідним?

7) У чому полягає сутність застосування конструкції **else if**?

8) Що таке умовний вираз?

9) Як у вигляді схеми алгоритму відображається складений оператор умови?

10) Як за допомогою одного оператора **if-else** перевірити декілька умов?

11) У якому випадку є необхідність у створенні сходинкової структури **if-else**?

12) Для чого застосовується оператор вибору?

13) У чому полягає різниця між складеним оператором умови та оператором вибору?

14) Яку функцію виконує гілка **default**?

15) Для чого застосовується оператор **break**?

16) Поясніть, як буде працювати оператор вибору без **break**?

17) Як виконати оголошення змінної у гілці **case**?

18) У чому полягає сенс застосування умовної операції?

19) Чим відрізняються **if-else** та умовна операція?

20) Як за допомогою "?:" знайти модуль числа?

5. ЦИКЛИ

Навчальною метою розділу є ознайомлення читача з циклічними програмними кодами.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– застосовувати прості оператори з перед- та з постумовами;

– реалізовувати складний оператор циклу;

– створювати вічні цикли на базі основних операторів циклів;

– використовувати оператори безумовного виходу та переходу до нової ітерації у циклах;

– створювати розгалуження та цикли на базі оператору **goto**.

5.1. Оператори циклів

Більшість явищ та процесів в житті і в промисловості носять циклічний характер – однакові дії виконуються N-у кількість разів. Причому, значення N може бути як близьким до нескінченності (наприклад, для неперервного виробництва), так і мати певне значення (наприклад, замовна партія продукції). В програмах також досить часто з'являється необхідність організації циклічного виконання будь-якої послідовності інструкцій та блоків.

Група умовних операторів, що слугує для формування циклічних повторень виконання програмного коду, називається операторами циклів. Вони поділяються на цикли з перед- і цикли з постумовою. У поточних реалізаціях мови С++ два оператори циклу з передумовою **while** та **for** і один оператор циклу з постумовою **do-while**.

5.1.1. Простий оператор циклу з передумовою while

while прийнято називати простим оператором циклу з передумовою. У схемах алгоритмів **while** відображається за допомогою блоку рішення, відрізняє його від **if-else** і **switch** сформована з блоків тіла циклу і ліній з'єднання петля, яка замикається на його вході (рис. 5.1).

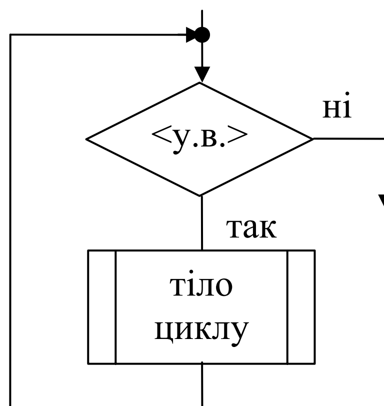

Рис. 5.1. Відображення оператора циклу з передумовою у схемах алгоритму

Відмінна риса циклів з передумовою полягає в тому, що умовний вираз, що визначає необхідно чи ні виконувати тіло циклу, стоїть перед вхідними в нього інструкціями.

Формат оператора **while** наведено нижче:

```
while(<умовний вираз> )
{
	<тіло циклу>
}
```

Умовний вираз виконує те саме завдання, що і для оператора **if-else**, тобто при його істинності інструкції, що формують тіло оператора виконуються, а при помилковості – ні. Проте, на відміну від операторів умови, після виконання інструкції тіла, в операторах циклів вихід з операторів не відбувається. Замість цього умовний вираз перевіряється знову. Таким чином, можна сказати, що тіло будь-якого циклу виконуватиметься по колу до тих пір, поки умовний вираз видає дійсний результат. Повне виконання інструкцій тіла прийнято називати ітерацією (рідше проходом). Цикл число ітерацій якого прагне до нескінченності, називають вічним. Починаючи вивчати операторів циклів, сенс використання вічних циклів розгледіти складно, проте він є.

Як і для складеного оператора умови, умовний вираз оператора **while** може набувати будь-якого значення. При цьому значущими є лише 0 і 1, а точніше 0 і всі інші числа, що компілятором асоціюються з 1.

Для формування вічного циклу на основі **while** достатнім є поставити будь-яке константне значення відмінне від 0 у якості умовного виразу, в ідеальному випадку –1.

Використання циклів в програмах дозволяє вирішувати безліч всіляких завдань: математичні (факторіал, арифметичні і геометричні прогресії, ряди, інтеграли тощо), потокові (тимчасові затримки, виконання до виникнення події, програмні потоки) роботи з масивами (читання і запис масивів, структур, об'єктів, рядків, файлів, буферів тощо) і багато інших.

Розглянемо вживання оператора **while** на прикладі програми для обчислення факторіала числа:

```
#include <stdio.h>                  //підключення бібліотеки вводу/виводу

void main()                         //оголошення та визначення головної функції
{
	int a, N = 1;                   //оголошення та визначення цілих змінних
	printf("Enter the value: ");    //вивід запиту до користувача
	scanf("%i", &a);                //отримання значення від користувача
	while(a > 0)                    //виконувати доки змінна більша за 0
		N *= a--;                   //помножити на поточне значення змінної
	printf("Factorial of the value is : % d", N);   //вивід результату
}
```

Для обчислення факторіала в програмі оголошуються дві змінні: a *– для набуття призначеного для користувача значення і* N *– для формування результату. Змінна* N *ініціалізується одиницею, оскільки при перемножуванні це значення не вплине на результат (за умовчуванням в оголошених змінних значення відмінні від* 0 *і* 1*). Запит користувачеві і читання введеного ним значення виконується за допомогою функцій* printf() *і* scanf(). *Цикл* **while** *виконується до тих пір, поки значення змінної* a *більше нуля. У тілі циклу поточне значення* N *домножається на поточне значення* a. *У цьому ж самому виразі* a *декрементується в постфіксній формі. Після виходу з циклу програма виводить користувачеві результат розрахунку. Для даної програми число ітерацій циклу* **while** *відповідатиме введеному користувачем цілочисельному значенню. Якщо буде введено негативне число або* 0*, то тіло циклу не буде виконано жодного разу.*

З прикладу видно, що як і **if-else**, оператор **while** може не мати фігурних дужок, що обмежують тіло, за умови, що в тілі міститиметься лише одна інструкція.

Розглянемо ще один приклад – програму, що виконує вивід на консоль таблиці ASCII кодів:

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

void main()                           //оголошення та визначення головної функції
{
        int i;                        //оголошення цілої змінної
        i = 0;          //визначення змінної за допомогою  початкового значення
        while(i < 256)                //виконується доки змінна є меншою за 256
        {
//вивести номер символу та символ
                printf("%0.3i = [% c]\r\n", i, i);
                i++;                  //інкрементувати змінну
        }
}
```

Для роботи з консоллю підключено бібліотеку форматного вводу/виводу. У головній функції виконується оголошення змінної лічильника. Ця змінна застосовується у циклі з передумовою для виводу на консоль таблиці ASCII *кодів. Цикл продовжується доки змінна не набуває значення* 256.

5.1.2. Простий оператор циклу з постумовою do-while

Оператор циклу **do-while** відноситься до циклів з постумовою. Його використання у програмі відповідає наведеному нижче формату:

```
do
{
	<тіло циклу>
}
while(<умовний вираз>);
```

У схемах алгоритмів оператор **do-while** теж реалізується на основі блоку рішення, але петля, що сформована з блоків тіла циклу і ліній з'єднання, формується перед ним (рис. 5.2).

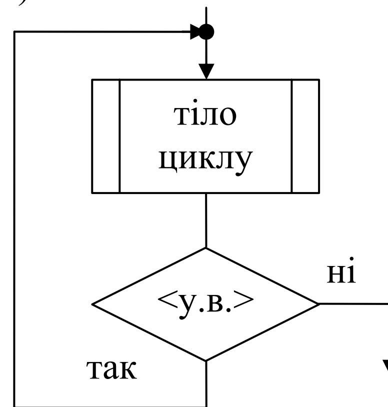

Рис. 5.2. Відображення оператора циклу з постумовою у схемах алгоритму

Особливістю циклів з постумовою є те, що їх тіло, на відміну від циклів з передумовою, виконується щонайменше одного разу. Це обумовлено тим, що перевірка умовного виразу здійснюється наприкінці оператора.

Оператор **do-while** є єдиним, після якого необхідно по синтаксисом C++ ставити ";", що пов'язано з особливостями роботи компілятора.

Як і для інших операторів у ролі умовного виразу може бути використано будь-який вираз, змінну, константу чи функцію, але логічний сенс підказує, що лише вирази з бінарним булевим результатом дозволяють розгалужувати програму.

Константне значення "1" замість умовного виразу приведе до вічного виконання циклу **do-while**. При значенні "0" цикл буде зведено лише до однієї ітерації, що зробить використання оператору **do-while** взагалі зайвим.

Таким чином, оператор циклу з постумовою зручно застосовувати для вирішення задач, у яких умовний вираз тим чи іншим чином залежить від програмного коду, що реалізується у тілі.

Розглянемо приклад використання **do-while** для пошуку значення факторіала введеного користувачем числа.

```
#include <stdio.h>                //підключення бібліотеки вводу/виводу

void main()                       //оголошення та визначення головної функції
{
	int N;                        //оголошення змінної для вводу числа
	int P=1;//оголошення та визначення змінної для розрахунку факторіала
```

```
	printf("N! will be calculated \r\n");	//вивід константного рядка
	printf("N value: ");			//вивід константного рядка
	scanf("%i", &N);			//отримання значення від користувача
	if(N>0)					//якщо введене число більше за 0
	{
		printf("%i! =", N);		//вивід на консоль початку виразу
		do				//цикл з постумовою
		{
			P*=N--;		//помноження на поточне значення
		}
		while(N>1);			//виконувати доки значення є більшим за 1
		printf("%i", P);		//вивід результату на консоль
	}
	else if(N==0)				//якщо число дорівнює 0
		printf("0!=1");		//вивід константного рядка
	else					//якщо число є від'ємним
		printf("Not valid N value"); //вивід константного рядка
}
```

Програма повідомляє користувача про своє призначення і запитує ввід значення N. *Факторіал* N *розраховується завдяки множенню поточного для ітерації циклу значення у змінній* P *на значення* N, *що покроково декрементується. Цикл закінчується у той час, коли* N *зменшиться до* 1. *Захист від хибного вводу не дозволяє розраховувати факторіал для від'ємних значень.*

5.1.3. Складний оператор циклу з передумовою for

Оператор циклу **for** також як і **while**, реалізує цикл з передумовою, але на відміну від нього є складним, що обумовлено особливістю його формату, а не важкості при застосуванні. Використання **for** у програмі відповідає нижче наведеному формату:

```
for(<початкове значення лічильника>;<умовний вираз>;
		<правило зміни значення лічильника>)
{
	<тіло циклу>
}
```

Лічильником для циклу **for** є змінна, значення якої дозволяє задавати кількість ітерацій циклу **for**. Така змінна може бути оголошена при вказуванні початкового значення лічильника чи може бути використана з попереднього коду програми. Умовний вираз при класичному використанні пов'язано з лічильником, але це не є обов'язковим. Перевірка умови виконується кожного разу перед ітерацією. Правило зміни задає, як змінюється значення лічильника при переході до наступної ітерації.

У схемах алгоритмів цикл **for** реалізується на основі блоку модифікації (рис. 5.3).

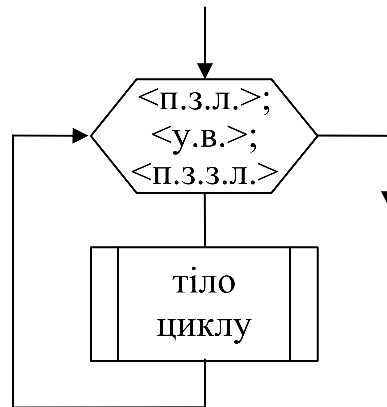

Рис. 5.3. Відображення складного оператора циклу у схемах алгоритму

Складним оператор циклу **for** називають завдяки тому, що в його параметрах задано ті вирази, що для інших операторів треба реалізовувати окремо у коді програми.

Також, як і для вище описаних **while** і **do-while**, оператор **for** має форму запису, що приводить до його вічного виконання:

```
for(;;)
{
	<тіло циклу>
}
```

Розглянемо приклад використання **for** для виводу на консоль таблиці ASCII кодів.

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

void main()                           //оголошення та визначення головної функції
{
	printf("ASCII list: \r\n");          //вивід константного рядка
	for(int c=0; c<=255; c++)            //цикл перебору значень від 0 до 255
		printf("[%3i]= '%c'\r\n", c, c);  //вивід номера символу та символу
}
```

Програма виконує перебір значень змінної с від 0 *до* 255 *включно, що відповідає повній нумерації* ASCII *кодів. Правило зміни с задається через інкремент змінної. Кожна ітерація циклу виводить на консоль строку, яка включає код у десятковій формі та сам символ.*

Треба зазначити, що вираз, вказаний у параметрі "початкове значення лічильника" виконується лише один раз, перед першою ітерацією. Параметр "правило зміни лічильника" застосовується лише після першої ітерації. Перед

кожною ітерацією перевіряється умова для підтвердження необхідності подальшого виконання циклу. Змінна, що була оголошена у параметрі "початкове значення лічильника" має видимість відповідно до самого оператора циклу і після його закінчення може бути використана у програмі до завершення поточного програмного блоку.

Роботу циклу **for**, як і інших операторів циклу, може бути перервано за допомогою оператору **break**.

Розглянемо використання оператора **for** для пошуку першого простого значення у заданому діапазоні.

```
#include <stdio.h>              //підключення бібліотеки вводу/виводу

void main()                     //оголошення та визначення головної функції
{
    int A, B;                   //оголошення змінних для вводу значень
//вивід константного рядка
    printf("Will be find first simple value from A to B \r\n");
    printf("A value:");         //вивід константного рядка
    scanf("%i", &A);            //отримання значення від користувача
    printf("B value:");         //вивід константного рядка
    scanf("%i", &B);            //отримання значення від користувача
    if(B>=A && A>0 && B>0)      //якщо діапазон значень зазначено правильно
    {
        for(int i=2; A<=B; i++) //цикл перебору значень дільника
        {
            if(i<A)             //якщо дільник є меншим за значення
            {
                if(A%i==0)      //якщо при діленні немає залишку
                {
                    A++;        //інкрементувати значення
                    i=1;        //задати нове значення дільника
                }
            }
            else                //якщо дільник дорівнює значенню
                break;          //вийти з оператора
        }
        if(A<=B)                //якщо просте значення з діапазону
            printf("%i is simple \r\n", i);    //вивід результату на консоль
        else                    //якщо просте значення не з діапазону
    //вивід константного рядка
            printf("No simple values from A to B \r\n");
    }
    else                        //якщо діапазон зазначено неправильно
        printf("No valid A or B \r\n");        //вивід константного рядка
}
```

Програма запитує у користувача ввід значень для змінних меж діапазону пошуку. Потім запускається цикл першого простого значення. Лічильник і *використовується для перебору значень від* 2 *до* A. *При кожній з ітерацій поточне значення* A *ділиться на поточне* і *через операцію* %. *Якщо результат виразу дорівнює* 0, *то число вважається невідповідним до простих і значення* A *змінюється через інкремент, а в* і *записується* 1. *У разі, коли лічильник досяг значення* A, *то поточне число є простим і цикл переривається за допомогою оператора* **break**. *При відсутності простих чисел у інтервалі від* A *до* B *цикл буде виконуватися до тих пір поки поточне значення змінної* A *буде меншим або таким, що дорівнює* B. *Захист від хибного вводу реалізовано завдяки перевірці змінних меж інтервалу на умови* A<B *та їх відмінність від* 0.

5.2. Оператори break та continue

Як вже було зазначено раніше, оператор **break** використовується для безумовного переривання роботи операторів **switch**, **while**, **do-while** та **for**. Принцип його дії полягає у наступному: протягом виконання програми при реалізації **break** здійснюється перехід на перший вираз, що йде за оператором, що переривається.

Оператор continue використовується не набагато рідше, ніж **break**, його функція полягає у перериванні поточної ітерації циклу і переходу до наступної. Реалізація оператору continue у тілі **if-else** або **switch** не призводить до помилки, але не несе сенсу.

Розглянемо приклад використання оператора continue у програмі, що виводить на консоль всі парні числа від 1 до 100 включно.

```
#include <stdio.h>              //підключення бібліотеки вводу/виводу

void main()                     //оголошення та визначення головної функції
{
	for(int i=1; i<=100; i++)//цикл перебору значень від 1 до 100
	{
		if(i%2==1)          //якщо значення парне
			continue;   //перейти до наступної ітерації
		printf("%i ", i);   //вивід значення на консоль
	}
}
```

Програма перебирає значення від 1 *до* 100 *за допомогою лічильника і оператору циклу* **for**. *Якщо поточне значення не є парним, то виконується перехід до наступної ітерації. В іншому випадку число виводиться на консоль.*

5.3. Оператор goto

Розгалуження програмного коду може бути реалізовано не лише на основі наведених вище операторів умови та циклів. Схожу функцію виконує оператор безумовного переходу **goto**. У схемах алгоритмів реалізація **goto**

виконується через зациклення за допомогою лінії зв'язку, що іноді пов'язана з блоком рішення. Формат його використання у програмі наведено нижче:

goto <*ім'я мітки*>;

Цей оператор може бути застосовано як для реалізації умовного розгалуження, так і для зациклення.

Міткою може бути використано будь-яке ім'я, що ще не було застосовано у програмі для інших цілей. Мітка ставиться з початку рядка, не зважаючи на форматування програмного коду, і відокремлюється від виразу рядка двоточкою. Формат використання мітки зазначено нижче:

<*ім'я мітки*> : <*вираз у програмі*>;

Розглянемо приклад використання оператору **goto** для розрахунку суми значень в інтервалі від 1 до N з кроком 1.

```
#include <stdio.h>                  //підключення бібліотеки вводу/виводу

void main()                         //оголошення та визначення головної функції
{
        char ch;      //оголошення змінної для вводу значення виходу/продовження
        int N;              //оголошення змінної для вводу значення для розрахунку
        int Sum;                    //оголошення змінної для розрахунку суми
//вивід константного рядка
        printf("Will summed values from 1 to N \r\n");
a:      Sum=0;                      //попереднє визначення змінної
        printf("N value:");         //вивід константного рядка
        scanf("%i", &N);            //отримання значення від користувача
        if(N>0)                     //якщо значення є більшим за 0
                goto b;             //перехід на мітку b
        else                        //інакше – значення є меншим за 0
                goto e;             //перехід на мітку  e
b:      Sum+=N--;                   //додавання до суми поточного значення
        if(N>0)                     //якщо значення є більшим за 0
                goto b;             //перехід на мітку b
        else                        //інакше – значення є меншим за 0
                goto c;             //перехід на мітку  c
c:      printf("Sum=%i \r\n", Sum); //вивід результату на консоль
d:      printf("1 for Quit, 0 for Continue: "); //вивід запиту до користувача
        scanf("%i", &ch);           //отримання вибору користувача
        if(ch)                      //якщо значення змінної – 1
                goto f;             //перехід на мітку f
        else                        //інакше, якщо інше значення
                goto a;             //перехід на мітку a
```

```
e:      printf("N must have positive value"); //вивід константного рядка
        goto d;                               //перехід на мітку d
f:      ;                                     //кінець розрахунків
}
```

Программа запитує у користувача значення для змінної N*, після чого, покроково зменшуючи його за допомогою декременту, додає поточне значення до змінної* Sum*, яку спочатку було визначено значенням* 0*. Цикл виконується доки значення* N *є більшим за* 0*. Після розрахунку результат виводиться на консоль, а користувачу ставиться запит на продовження роботи. При виборі* 0 *програма повертається до запиту значення* N*. Значення* 1 *призводить до закінчення програми. Всі умови у програмі реалізовані за допомогою умовної операції, яка була розглянута раніше.*

Використання оператору **goto** є дуже рідким, бо вважається "хибним тоном" при написанні програм. Таке відношення пов'язано з тим, що застосування **goto** дуже ускладнює розуміння програмного коду і може призвести до появи логічних помилок. Це важко визначити.

Висновки

У даному розділи було розглянуто наступні основні питання:
– простий оператор циклу з передумовою while;
– простий оператор циклу з постумовою do-while;
– складний оператор циклу з передумовою for;
– оператори break та continue;
– оператор goto.

Контрольні питання

1) Поясніть зміст роботи простого оператора циклу.

2) У якому випадку тіло простого оператору циклу можна не брати в фігурні дужки?

3) Як реалізується вічний цикл на основі оператору **while**?

4) Як складний оператор циклу відображається у схемі алгоритму?

5) Порівняйте прості оператори циклу з перед- та постумовами.

6) Наведіть формат застосування простого циклу з передумовою.

7) Яка особливість характеризує всі цикли з постумовою?

8) У чому полягає особливість формату оператора циклу з постумовою?

9) Як простий оператор циклу з постумовою відображається у схемі алгоритму?

10) Чому оператор **for** називають складним?

11) Для чого у складному операторі циклу застосовується крок зміни значення лічильника?

12) Як реалізується вічний цикл на основі оператора **for**?

13) Як складний оператор циклу з передумовою відображається у схемі алгоритму?

14) Порівняйте оператори циклу **for** та **while**.

15) Поясніть як працює оператор **continue**?

16) Для чого у циклах застосовується оператор **break**?

17) Який формат застосування має оператор безумовного переходу?

18) Що називають міткою?

19) Як на операторі безумовного переходу реалізувати циклічний код?

20) Чому застосування **goto** вважають "хибним тоном" при написанні програм?

6. ЗМІННІ-ПОКАЖЧИКИ

Навчальною метою розділу є ознайомлення читача із застосуванням змінних-покажчиків для збереження адрес пам'яті та роботи з даними, що знаходяться у відповідних комірках.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– отримувати адреси змінних програми;
– створювати змінні-покажчики;
– працювати зі значеннями через змінні-покажчики;
– виконувати динамічне виділення пам'яті;
– вивільнювати пам'ять.

6.1. Доступ до пам'яті за допомогою змінних-покажчиків

Як вже було зазначено, тип змінної визначає кількість байтів пам'яті, що виділяється для зберігання значень, якими програма оперує в ході свого виконання. У свою чергу, ім'я змінної виступає в ролі засобу адресації до виділеної під значення пам'яті. Проте змінна не є явною адресою, сенс її оголошення полягає в створенні зрозумілої для програміста псевдоадреси, за якою стає зручним виконувати операції зі значенням змінної.

Явною адресою для будь-якої змінної є цифрове значення, яке виступає в ролі номера першого елементу пам'яті з блоку, займаного нею. Адресу будь-якої змінної можна отримати за допомогою оператора "&". Ця операція так і називається – отримання адреси. Вивести адресу на консоль через функцію форматного виводу можна із застосуванням специфікатора формату – "%p".

Наприклад:

```
int k;              // оголошення цілої змінної k
printf("%p", &k);  //вивід адреси змінної  k на консоль
```

Отримана адреса може під час кожного нового запуску програми мати різне значення (специфікатор формату виводить значення у шістнадцятковому вигляді). Це пов'язано з тим, що при появі нового процесу у системі йому виділяється у якості ресурсу адресний простір у фізичній пам'яті.

Дії зі значеннями змінних можна проводити як через її ім'я, так і через явну адресу. Оскільки адреса виділеної для зберігання даних пам'яті в рамках програмного коду також може виступати в ролі значення, це призводить до необхідності створення окремого типу змінних. Змінні, що зберігають адреси пам'яті, називаються змінними-покажчиками. Проте тип даних для них явно не визначено.

Формат оголошення змінної-покажчика наведено нижче:

<*тип даних*> *< *ім'я змінної-покажчика*>;

Всі змінні-покажчики містять значення, які є адресами пам'яті, але при цьому не можуть мати окремий тип даних, оскільки важливою є не тільки

адреса першої комірки, але і їх кількість. Для зберігання даних може бути залучено різна кількість комірок. Таким чином, при оголошенні змінної-покажчика основну роль грає її ім'я. Символ "*" відноситься до імені та застосовується для того, щоб визначити змінну як покажчик на тип даних, що характеризує значення за адресою.

Наприклад:

```
int k;            //оголошення цілої змінної k
int *p = &k;      //визначення змінної-покажчика адресою k
```

Важливо розуміти, що використовувати змінну-покажчик можна тільки після її визначення, оскільки значення, що зберігається в ній за умовчуванням після оголошення не є адресою комірки виділеної процесу пам'яті. І використання такого значення може призвести до системних збоїв або навіть краху сеансу операційної системи.

Для змінних-покажчиків, як і для звичайних змінних, існує операція переприсвоювання, але на неї накладено деяке обмеження. Так, значення змінної-покажчика може бути переприсвоєне лише змінній-покажчику, що вказує на однойменний тип даних або на тип даних **void**. Необхідність такого обмеження цілком виправдана, бо в іншому випадку при доступі до даних результат операції міг би бути невизначеним. Проте переприсвоювання значень змінній-покажчику на тип даних **void** не відповідає цьому правилу. Виключення пов'язано з тим, що **void** є універсальним – займає в пам'яті максимальну кількість комірок, що є більшою або такою, що дорівнює будь-якому іншому типу даних, і при цьому може містити значення будь-якого типу. Ця особливість у сумісності з операцією явного перевизначення типу даних дозволяє **void** виступати в ролі контейнера при передачі будь-яких значень.

Наприклад:

```
int i;            //оголошення цілої змінної
float f;          //оголошення змінної з плаваючою точкою

int *pi;          //оголошення змінної-покажчика на цілий  тип даних
float *pf; //оголошення змінної-покажчика на тип даних з плаваючою точкою
void *pv;         //оголошення змінної-покажчика на порожній тип даних

pi = &i;          //присвоєння pi адреси змінної i
//pi = &f;        //неприпустима операція – типи несумісні
//pf = &i;        //неприпустима операція – типи несумісні
pf = &f;          //присвоєння pf адреси змінної f
pv = &i;          //присвоєння pv адреси змінної i
pv = &f;          //присвоєння pv адреси змінної f
```

Однак змінні-покажчики слугують не тільки для зберігання адрес інших змінних. Як вже було сказано вище, будь-яка дія із значенням змінної може

бути виконана і через змінну-покажчик. Операція доступу до значення через змінну-покажчик називається розіменуванням. Вона заснована на застосуванні оператора "*", який є унарним і записується у коді перед ім'ям змінної-покажчика.

Наприклад:

```
int i = 10;        //оголошення та визначення цілої змінної
int k;             //оголошення цілої змінної
//оголошення та визначення змінної-покажчика на цілий тип даних
int *pi = &i;
k = *pi;           //визначення змінної k значенням, на яке вказує адреса у pi
//визначення змінної, адресу якої містить змінна-покажчик pi
*pi = 15;
```

Після виконання цього програмного коду в змінну k буде записано значення 10, а в змінну i – значення 15.

Важливо розуміти, що при відповідності формату операції розіменування змінної-покажчика форматові її оголошення, вони мають абсолютно різний сенс. Використання символу "*" для оголошення визначає вказівку до компілятора про те, що наступна змінна є покажчиком, а для розіменування – про формат прямого звернення до пам'яті через явну адресу.

Всі наведені вище дії можна було б виконати і без використання змінних-покажчиків. Причому їх застосування у простих програмах скоріше ускладнює і подовжує код. Але у випадках, що наведено нижче, змінні-покажчики є незамінними:

- створення масивів довільної величини (динамічних масивів);
- передача масивів у функцію та доступ до їх елементів;
- отримання з функції декількох значень через її аргументи;
- створення пов'язаних списків.

6.2. Динамічне виділення пам'яті

Для виділення і вивільнення пам'яті в ході виконання програми використовуються службові слова **new** і **delete**. Цей підхід потрібен, коли при виконанні програми виникає необхідність зберегти які-небудь дані, а змінну для цієї операції не було передбачено, або при використанні масивів змінної довжини. Операція виділення пам'яті виконується для змінної-покажчика, з якою і буде пов'язано комірки для динамічного зберігання даних. Формат цієї операції зазначено нижче:

```
<тип даних> *<ім'я змінної-покажчика> = new <тип даних>;
```

При виділенні пам'яті змінна-покажчик визначається адресою першої з ряду комірок, який виділено для зберігання значень відповідного типу даних. Такі змінні називають динамічними. Вони існують доти, доки пам'ять не буде вивільнено. Вивільнення виконується за наступним форматом:

delete <*ім'я змінної-покажчика*>;

При цьому раніше виділена під значення зміною пам'ять вивільнюється, що дозволяє системі використовувати її на свій розсуд.

Наприклад:

```
//оголошення змінної-покажчика на цілий тип даних та його визначення
int *pi = new int;
*pi = 15;              //запис у  виділену пам'ять значення 15
printf("%i", *pi);  //вивід значення на консоль
delete pi;             //вивільнення виділеної пам'яті
```

Після звільнення пам'яті записані до неї дані втрачаються.

Більш докладно змінні-вкзівники та динамічне виділення пам'яті будуть розглянуті у наступних темах.

Висновки

У даному розділи було розглянуто наступні основні питання:

– доступ до пам'яті за допомогою змінних-покажчиків;

– динамічне виділення пам'яті.

Контрольні питання

1) Що називають змінною-покажчиком?

2) Чому змінні-покажчики не мають свого окремого типу даних?

3) У чому сенс застосування типу даних void при роботі зі змінними-покажчиками?

4) Як виконується оголошення змінної-покажчика?

5) Які операції пов'язано з застосуванням змінних-покажчиків?

6) У яких випадках змінні-покажчики є незамінними?

7) Що називають динамічною змінною?

8) Як виконується динамічне виділення пам'яті?

9) Як вивільнити динамічно виділену пам'ять?

10) Для чого використовується операція розіменування змінної-покажчика?

7. МАСИВИ

Навчальною метою розділу є ознайомлення читача із застосуванням масивів для збереження великих об'ємів даних одного типу.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– виконувати оголошення масивів;
– працювати з елементами масиву;
– визначати масив при оголошенні;
– застосовувати змінні-покажчики для роботи з масивами;
– створювати динамічні масиви;
– вивільнювати пам'ять, що зайнято під динамічні масиви.

7.1. Робота з великими об'ємами даних

Всі програми, системні сервіси чи ігри, працюють з даними. Їх кількість під час роботи деяких програм може сягати як мегабайтів, так і терабайтів. Тому дуже важливою є така побудова програмного коду, що дозволяє опрацьовувати великі об'єми значень без впливу на розміри самої програми. Для цього застосовується буферний підхід. Для даних виділяється об'єм пам'яті, якого вистачає для тимчасового зберігання кількості значень, що обробляються програмою за визначений час роботи. Оброблені дані перезаписуються тими, що ще потребують опрацювання. Базисом для цього підходу є те, що з якою б кількістю даних не працювала програма, вона виконує їх обробку поступово, а не за один раз.

Звичайних змінних для реалізації буферу замало, бо навіть при поступовій обробці необхідно було б створювати їх сотні. Зрозуміло, що така кількість змінних значно б ускладнила роботу програміста, – кожна змінна повинна була б мати своє унікальне ім'я і це б зробило застосування структурного підходу майже зовсім неможливим. Тому для зберігання та опрацювання великих об'ємів даних у мовах програмування використовуються масиви.

7.2. Масиви

Масив – це сукупність змінних одного типу даних, що поєднані під одним ім'ям.

Змінні, що входять до массиву, називають його елементами. Кожна з них має свій унікальний порядковий номер, що надає можливість роботи з нею окремо від інших. Такий номер прийнято називати індексом масиву.

Для будь-якого масиву нумерація індексів починається з 0. Таким чином, для масиву, що містить 10 елементів, індекси будуть знаходитись у діапазоні від 0 до N-1, де N – кількість елементів.

Розрізняють одновимірні і багатовимірні масиви. Серед багатовимірних для зберігання даних найчастіше використовуються двовимірні та тривимірні, бо їх уявлення є найбільш зрозумілим.

Одновимірні масиви зазвичай називають векторами значень, а двовимірні – матрицями.

7.2.1. Оголошення масиву

Формат оголошення масиву у програмі наведений нижче:

<*тип даних*><*ім'я масиву*>[<*1-а розмірність*>]...[<*N-а розмірність*>];

Розмірності масиву – це константні цілі значення, що задають число елементів. При наявності декількох розмірностей кількість елементів відповідає їх добутку.

Наприклад:

```
//оголошення одновимірного цілого масиву з 10 елементів
int Mas[10];
//оголошення двовимірного символьного масиву з 25 елементів
char Matrix[5][5];
//оголошення тривимірного дійсного масиву з 30 елементів
float Matrix[2][3][5];
```

Елементи масиву можна як і звичайні змінні визначати при оголошенні і в програмі. При оголошенні для визначення елементів масиву застосовується наступний формат:

<*тип даних*><*ім'я масиву*>[< *розмірність*>]={<список значень>};

Всі масиви у пам'яті представляють собою ряд комірок для зберігання даних, тому ініціалізація багатовимірних масивів також може бути виконана без поділення даних за размірностями. Але це знижує читаність коду.

При визначенні масиву за розмірностями під час оголошення, елементи кожного вектору розміщуються через кому в окремих фігурних дужках. Окрім підвищення читаності, таке визначення дозволяє також виконати часткову ініціалізацію. Елементи, значення для яких не вказано, будуть ініціалізовані нулями.

Якщо визначення відбувається при оголошенні, то значення розмірності одновимірного масиву можна не вказувати, компілятором буде створено масив з кількістю елементів, що відповідає кількості даних для ініціалізації.

Наприклад:

```
//оголошення та визначення одновимірного масиву
int Vector0[7]={3,6,12,42,8,-1,0};
//оголошення та визначення одновимірного масиву без вказування размірностей
int Vector1[]={3,6,12,42,8,-1,0};

//оголошення та визначення двовимірного масиву
int Matrix0[3][2]={{2,6},{1,5},{7,3}};
//оголошення та визначення у рядок двовимірного масиву
int Matrix1[3][2]={2,6,1,5,7,3};
```

```
//оголошення та часткове визначення двовимірного масиву
int Matrix2[3][2]={{2},{1},{7}};

//оголошення та визначення тривимірного масиву
int Mas0[2][2][2]={{{1,2},{3,4}},{{5,6},{7,8}}};
//оголошення та визначення у рядок тривимірного масиву
int Mas1[2][2][2]={1,2,3,4,5,6,7,8};
//оголошення та часткове визначення тривимірного масиву
int Mas2[2][2][2]={{{1},{3}}};
```

При виводі отриманих масивів на консоль отримаємо наступний результат:

```
Vector0:
3   6   12   42   8   -1   0

Vector1:
3   6   12   42   8   -1   0

Matrix0:
2   6
1   5
7   3

Matrix1:
2   6
1   5
7   3

Matrix2:
2   0
1   0
7   0

Mas0:
    1   2
3   4

    5   6
7   8
```

Mas1:
  1 2
3 4

  5 6
7 8

Mas2:
  1 0
3 0

  0 0
0 0

Для одновимірного, двовимірного та тривимірного масивів перші два визначення ініціалізують елементи однаково. При третьому визначенні двовимірного та тривимірного масивів значення, що не вказані, за умовчуванням сприйнято як нулі.

7.2.2. Робота з елементами одновимірного масиву

У програмному коді визначення елементів масиву відбувається відповідно до звичайних змінних. Доступ до окремих елементів виконується із застосуванням імені масиву та унікального індексу.

Так, наприклад, для визначення п'ятого елемента цілого масиву Mas[20] та виводу його значення на консоль може бути застосовано наступний код:

```
Mas[4]=31;                          //визначення п'ятого елемента масиву
printf("Mas[4]=%i\r\n",Mas[4]);     //вивід на консоль значення елемента
```

Якщо визначення всіх елементів масиву виконано у програмі за допомогою оператору циклу та правила для формування значень, то такий код називають автоматичною ініціалізацією.

Наприклад, для заповнення масиву Arr[10] значеннями від 1 до 10 може бути задіяно наступний цикл:

```
for (int i=0; i<10; i++)   //цикл перебору значень індексів від 0 до 9
{
        Arr[i]=i;          //визначення елемента масиву
}
```

Схожий підхід може бути використано і для ініціалізації масиву значеннями користувача:

```
for (int i=0; i<10; i++)          // цикл перебору значень індексів від 0 до 9
{
```

```
//вивід на консоль запиту до користувача
        printf("Enter value for Arr[%i] element:", i);
        scanf("%i", &Arr[i]);      // вичитування значення елемента масиву
        printf("\r\n");            //перехід на новий рядок
}
```

Операції з елементами масиву проводяться через ім'я масиву і індекс елемента також, як і при ініціалізації.

Розглянемо приклад програми, що виконує розрахунок виразу $y=x^2$ для цілих значень в діапазоні від 1 до 10, записує результати обчислень в масив M[10] та виводить його у зворотній послідовності:

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

void main()                           //визначення головної функції
{
        int M[10];                    //оголошення цілого масиву з 10 елементів
        int i=0;                 //оголошення і визначення цілої змінної лічильника
        while((++i) <= 10)            //цикл перебору значень від 1 до 10
                M[i-1]=i*i;           //визначення елемента масиву
        for(i=9; i>=0; i--)           //цикл виводу елементів масиву з 9 по 0
                printf("%i\t", M[i]); //вивід значення поточного елемента
}
```

Директива **include** *підключає бібліотеку для роботи з форматними функціями вводу/виводу. У головній функції виконується оголошення масиву* M *з розмірністю для збереження десяти цілих значень. Потім визначається значенням* 0 *змінна-лічильник. За допомогою циклу з передумовою масив автоматично ініціалізується значеннями квадратів чисел від* 1 *до* 10. *Визначений масив із застосуванням складного циклу з передумовою виводиться на консоль у зворотній послідовності відповідно до порядку його елементів.*

7.2.3. Робота з елементами багатовимірного масиву

При роботі з багатовимірними масивами у програмі для доступу до його елементів необхідно зазначати індекси за всіма розмірностями. Так, наприклад, для визначення елемента матриці Mtrx[7][8], що знаходиться на перетині п'ятого рядка та четвертого стовпця, і виводу його значення на консоль можна скористатися наступним кодом:

```
Mtrx[4][3]=124;   //визначення елемента п'ятого рядка четвертого стовпця
printf("Matrix[4][3]=%i", Mtrx[4][3]); //вивід значення елемента на консоль
```

Перебір елементів багатовимірних масивів також може бути виконано за допомогою операторів циклу.

Розглянемо приклад програми для отримання матриці C, як добутку матриць A та B розмірністю 3x3:

```
#include <stdio.h>              //підключення бібліотеки вводу/виводу

void main()                     //оголошення та визначення головної функції
{
	int A[3][3];                //оголошення матриці A
	int B[3][3];                //оголошення матриці B
	int C[3][3];                //оголошення матриці C
	int i,j;                    //оголошення змінних лічильників для циклів
//вивід на консоль запиту до користувача
	printf("Enter values for A and B matrix:\r\n");
	for(i=0; i<3; i++)          //цикл перебору рядків матриці A
		for(j=0; j<3; j++)  //цикл перебору стовпців матриці A
		{
			printf("A[%i][%i]=",i,j); //вивід запросу до користувача
			scanf("%i",&A[i][j]);     //отримання значення елемента
		}
	for(i=0; i<3; i++)          //цикл перебору рядків матриці B
		for(j=0; j<3; j++)  //цикл перебору стовпців матриці B
		{
			printf("B[%i][%i]=",i,j); //вивід запросу до користувача
			scanf("%i",&B[i][j]);     //отримання значення елемента
		}
	for(i=0; i<3; i++)          //цикл перебору рядків матриці C
		for(j=0; j<3; j++)  //цикл перебору стовпців матриці C
//розрахунок значення поточного елемента матриці C
			C[i][j]=A[i][0]*B[0][j]+A[i][1]*B[1][j]+A[i][2]*B[2][j];
	for(i=0; i<3; i++)          //цикл перебору рядків матриць A, B, C
	{
		for(j=0; j<3; j++)          //цикл перебору стовпців матриці A
			printf("%5.i",A[i][j]); //вивід значення поточного елемента
		printf("  ");                       //вивід двох символів пробілу
		printf((i==1) ? ("X") : (" "));     //вивід символу 'X' чи пробілу
		for(j=0; j<3; j++)          //цикл перебору стовпців матриці B
			printf("%5.i",B[i][j]); //вивід значення поточного елемента
		printf("  ");                       //вивід двох символів пробілу
		printf((i==1) ? ("=") : (" "));     //вивід символу '=' чи пробілу
		for(j=0; j<3; j++)          //цикл перебору стовпців матриці C
			printf("%5.i",C[i][j]); //вивід значення поточного елемента
		printf("\r\n");             //перехід на новий рядок
	}
}
```

Директива **include** *підключає бібліотеку для роботи з форматними функціями вводу/виводу. У головній функції виконується оголошення двовимірних цілих масивів* A, B, C *з розмірностями 3x3. Потім оголошуються змінні лічильників циклів. Після виводу на консоль запиту на ввід значень елементів матриць, за допомогою складних циклів з передумовою масиви* A *та* B *заповнюються даними користувача. Наступний складний цикл виконує розрахунок значень елементів матриці* C *шляхом помноження відповідних рядка та стовпця. Останній цикл застосовано для виводу результатів розрахунку у вигляді формули* A x B = C.

При вводі значень, що відповідають нижче наведеним матрицям A та B,

```
A           B
1  2  3     1  2  3
4  5  6     4  5  6
7  8  9     7  8  9   ,
```

результат роботи програми буде наступним:

```
 1  2  3       1  2  3      30   36   42
 4  5  6   x   4  5  6  =   66   81   96
 7  8  9       7  8  9     102  126  150
```

7.3. Робота з елементами масивів за допомогою змінних-покажчиків

Застосування змінних-покажчиків для роботи з масивами найбільш чітко розкриває причину явного вказування типу даних при їх оголошенні. Так, для символьного масиву з 10 елементів об'єм виділеної пам'яті відповідає 10 байтам, а для цілого масиву такої ж розмірності – 20 байтам (відповідно до того, що на кожний елемент цілого типу даних виділяється по 2 байти, а на елементи символьного – по 1 байту). При цьому, якщо змінна-покажчик не буде включати інформацію про тип даних значень, адреси яких можуть бути в неї записані, то цілий масив може бути інтерпритовано як символьний з 20-ти елементів (при адресуванні до кожного байта), чи символьний як цілий з 5-ти елементів (при адресуванні до групи з двох байтів). Щоб запобігти такому непорозуміння, застосовується тип даних значень при оголошенні змінних-покажчиків.

Тип даних, що зазначено для змінної-покажчика визначає крок, на який треба змінювати адресу при інкрементуванні чи дикрементуванні. Таким чином, адреса значень символьного типу змінюється на 1, а адреса цілого типу – на 4.

Для роботи з масивом через змінну-покажчик достатньо скористатися його ім'ям. У кожного масиву ім'я є константним покажчиком на виділений під його дані об'єм пам'яті. Таким чином, вирази для цілого масива

```
//оголошення та визначення цілого масиву з п'яти елементів
int Arr[5]={4,1,7,2,8};
```

є еквівалентними:

```
int *pi = Arr;          //отримання адреси масиву
int *pi = &Arr[0];      //отримання адреси масиву
```

Це пов'язано з тим, що адреса самого масиву та адреса його першого елемента є одне й те саме.

Таким чином, для перебору елементів за допомогою змінних-покажчиків можна використовувати ім'я масиву.

Наприклад:

```
int Arr[5] = {3,6,1,7,2};         //оголошення та визначення масиву Arr

for (int i=0; i<5; i++)           //цикл перебору елементів массиву
{
      printf("%i\t", *(Arr+i));  //вивід на консоль поточного елемента масиву
}
```

Важливо розуміти, що константна змінна-покажчик відрізняється від звичайної тим, що адресу в ній перезаписати неможливо. В іншому випадку у протягом роботи з масивом можна було б втратити зв'язок з пам'яттю, що було виділено під зберігання даних. Тому, для отримання адрес елементів масиву необхідно використовувати допоміжну змінну, що буде грати роль індексу і визначати крок між початком масиву та поточним елементом.

При застосуванні звичайної змінної-покажчика програмісту над зазначеною особливістю замислюватись немає потреби – операції з нею не створюють впливу на статично виділену пам'ять.

Таким чином, змінні-покажчики дозволяють відмовитись від використання індексів елементів масиву. Так, для наведеного вище прикладу під час застосування змінної-покажчика при переборі елементів за допомогою циклу можна застосовувати інкрементування:

```
int Arr[5] = {3,6,1,7,2}; //оголошення та визначення масиву Arr
int *pA = Arr;      //оголошення та визначення змінної-покажчика на масив
for (int i=0; i<5; i++)     //цикл перебору елементів масиву
{
      printf("%i\t", *(pA++)); //вивід на консоль поточного елемента масиву
}
```

7.4. Динамічні масиви

Розглянуті вище масиви мають константну розмірність, що може бути незручним в тому випадку, коли при написанні програми немає інформації про те, яку кількість даних одного типу необхідно буде зберігати. Якщо навмисно зазначити, що масив буде мати кількість елементів більшу, ніж може колись

знадобитися, то це призведе до марної трати пам'яті. В іншому випадку – масиву може не вистачити для вирішення задач користувача. Цього можна запобігти за допомогою застосування масивів змінної довжини, які називають динамічними. Їх створення базується на використання динамічного виділення пам'яті.

Формат створення динамічного масиву наступний:

<*тип даних*> *<*ім'я змінної-покажчика*> = **new** <*тип даних*>[<*розмірність*>];

При цьому виділяється область пам'яті, що відповідає кількості байтів, що необхідні для зберігання значень елементів масиву вказаної довжини. Розмірність динамічного масиву не є константою – у її якості може бути використано будь-яку цілу змінну чи константне значення.

Вивільнення пам'яті відбувається за допомогою оператора **delete**. Формат вказано нижче:

delete [] <*ім'я змінної-покажчика*>;

Виконувати вивільнення пам'яті можна лише тоді, коли змінна-покажчик була визначена, в іншому випадку ця операція призведе до помилки.

Розглянемо приклад програми, що розраховує відхилення кожного з значень масиву від максимального елемента. Масив має змінну довжину і визначається користувачем:

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

void main()                           //оголошення та визначення головної функції
{
        int r;              //оголошення змінної для визначення розміру масиву
        printf("Enter size of arrey: ");  //вивід запиту до користувача
        scanf("%i", &r);              //отримання значення розміру масиву
        int *pArr = new int[r];       //створення динамічного масиву
        for(int i=0; i<r; i++)        //цикл перебору елементів масиву
        {
                printf ("Enter Arr[%i]: ", i);    //запит на ввід значення
                scanf("%i", pArr+i);              //отримання значення
        }
        int max=*pArr;     //визначення змінної пошуку максимального значення

        for(i=1; i<r; i++)                      //цикл перебору елементів масиву
//якщо поточний елемент більше максимального
                if(*(pArr+i)>max)
                        max=*(pArr+i);          //перевизначити змінну
//вивід на консоль знайденого максимального значення
        printf("\r\nMAX value is %i\r\n", max);
```

```
//вивід на консоль константного рядка
        printf("Arrey values like a difference with MAX:\r\n");
//вивід на консоль символу фігурної дужки
        printf("{");
        for(i=0; i<r; i++)          //цикл перебору елементів масиву
                if(i!=r-1)          //якщо поточний елемент не останній
                        printf("%i,", max-*(pArr+i));   //вивести різницю і кому
                else                //якщо поточний елемент останній
                        printf("%i", max-*(pArr+i));    //вивести різницю
        printf ("}\r\n");           //вивід на консоль символа фігурної дужки
        delete [] pArr;             //вивільнення пам'яті
}
```

Для роботи з функціями форматного вводу/виводу підключено бібліотеку stdio.h. У головній функції виконується оголошення змінної для визначення розмірності динамічного масиву. Після її ініціалізації значенням користувача виділяється пам'ять для зберігання відповідної кількості даних. Потім за допомогою складного циклу з передумовою динамічний масив ініціалізується значеннями користувача. Після закінчення визначення виконується пошук максимального значення серед елементів масиву. Отримане значення виводиться на консоль. Останній складний цикл виконує розрахунок і вивід на консоль в рядок відхилень значень елементів від максимального.

Таким чином, застосування динамічних масивів створює можливість розробки універсальних програм, що впродовж своєї роботи використовують лише ту кількість пам'яті, що є необхідною для вирішення поставленого завдання.

7.5. Застосування масивів

Статичні та динамічні масиви використовуються у більшості програм, що розробляється, як для системи, так і для користувача. Масиви є однаково потрібними на високому та на низькому рівнях програмування. Їх застосування визначає основу для застосування механізму буферізації.

На базі масивів працюють наступні алгоритми та механізми:
- робота з матрицями;
- динамічні масиви;
- FIFO буферізація;
- відтворення стеків чи LIFO буферізація;
- алгоритми впорядкування;
- пов'язані списки;
- робота з графами;
- зберігання координат при роботі з графікою тощо.

Висновки

У даному розділи було розглянуто наступні основні питання:

– одномірні масиви;
– багатомірні масиви;
– динамічні масиви.

Контрольні питання

1) У чому полягає сенс застосування механізму буферізації?
2) Що називають масивом?
3) Дайте визначення елемента масиву?
4) Чим характеризуються окремі елементи масиву?
5) Що визначає розмірність масиву?
6) Які бувають масиви?
7) Як ще називають одновимірний та двовимірний масиви?
8) Як виконується визначення масиву при оголошенні?
9) Для чого може бути необхідним розділення значення при визначенні масиву під час оголошення?
10) Що називають автоматичною ініціалізацією масиву?
11) Як отримати доступ до сьомого елемента вектора?
12) Як у програмі працювати з елементами двовимірних масивів?
13) Яка особливість є у імені масиву?
14) Чому константна змінна-покажчик не може бути перезаписана?
15) Як отримати доступ до значення елемента масиву через змінну-покажчик?
16) Для чого потрібні динамічні масиви?
17) Як створюється динамічний масив?
18) Як вивільнити пам'ять, що виділено під динамічний масив?
19) У чому полягає необхідність застосування різних типів даних при оголошенні змінних-покажчиків?
20) Для реалізації яких підходів та алгоритмів можуть бути застосовані масиви?

8. СКЛАДЕНІ ТИПИ ДАНИХ

Навчальною метою розділу є ознайомлення читача з складеними типами даних мови C++.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– створювати структури;

– застосовувати ім'я структури для оголошення екземплярів;

– працювати з полями структури через її екземпляр;

– застосовувати змінні-покажчики при роботі з динамічними екземплярами структури;

– створювати та використовувати у програмі іменовані та безіменні поєднання;

– працювати з переліченнями.

8.1. Складені типи даних

Складеними у мовах програмування називають типи даних, що формуються програмістом на основі структур, поєднань чи перелічень. Як і звичайні типи даних, складені характеризують об'єм пам'яті, що виділяється при оголошенні змінних для збереження відповідних значень. Складеними типи даних називають у зв'язку з тим, що вони поєднують декілька простих типів даних.

8.1.1. Структури

Структура – це сукупність одного чи різних типів даних, що погруповано під єдиним ім'ям за загальним змістом.

Ім'я створеної структури у програмі має сутність складеного типу даних і може бути задіяне задля оголошення змінних, котрі прийнято називати екземплярами. Змінні, що включено до структури, є її членами чи полями.

Кожне поле у структурі має своє унікальне ім'я. Для оголошення структури використовується службове слово **struct**.

struct<*ім'я структури*>
{
        <*оголошення полів*>
} <*список екземплярів структури*>;

Список екземплярів структури може бути залишено порожнім. Оголошення структури без зазначення екземплярів не є для компілятора приводом виділення пам'яті під збереження значень. Пам'ять застосовується лише на екземпляри, причому на кожний новий окремо. Таким чином, поля кожного екземпляру структури є незалежними і можуть бути використані для збереження різних значень.

При програмуванні структура може мати декілька призначень: опис характеристик будь-якого предмета чи об'єкта, передача та отримання даних з функції, формування таблиць даних, створення пов'язаних списків тощо.

Доступ до полів статичного екземпляра з програми виконується через операцію ".".

Розглянемо приклад програми, що використовує структури для побудови таблиці реєстрації товарів одягу.

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

struct clothes                 //оголошення структури характеристик одягу
{
    char name[20];                    //оголошення поля назви одягу
    int size;                         //оголошення поля розміру одягу
    float price;                      //оголошення поля ціни одягу
};

void main()                           //оголошення та визначення головної функції
{
    char choice; //оголошення змінної для вводу значення виходу/продовження
    clothes c[50];                    //оголошення масиву екземплярів структури
    for(int i=0; i<50; i++)           //цикл перебору елементів масиву
    {
        printf("Enter name: ");   //вивід запиту до користувача
        scanf("%s", c[i].name); //отримання значення від користувача
        printf("Enter size: ");   //вивід запиту до користувача
        scanf("%i", &c[i].size);  //отримання значення від користувача
        printf("Enter price: ");  //вивід запиту до користувача
        scanf("%f", &c[i].price);//отримання значення від користувача
        printf("Continue (Y/N)?: ");     //вивід запиту до користувача
        scanf("%s", &choice);     //отримання значення від користувача
        if(choice=='N')           //якщо було введено символ 'N'
            break;                //вийти з оператора
    }
    printf("\r\n\r\n\r\n");           //зробити 3 переходи на новий рядок
    printf("Clothes table: \r\n");           //вивід константного рядка
    for(int j=0; j<=i; j++)     //цикл перебору елементів, що було заповнено
    {
        printf("%s\t", c[j].name);//вивід значення на консоль
        printf("%i\t", c[j].size);           //вивід значення на консоль
        printf("%.2f\t", c[j].price);        //вивід значення на консоль
        printf("\r\n");                      //перехід на новий рядок
    }
}
```

За допомогою директив підключаються бібліотеки роботи з форматним вводом/виводом. Виконується оголошення структури clothes *з трьома полями:* name*,* size *та* price*. У головній функції реалізовано масив, вихід з якого можна*

здійснити двома шляхами: обравши 'N' на запит чи заповнивши масив з п'ятдесяти екземплярів структури. Кожна ітерація покроково запитує у користувача значення для полів чергового елемента. Після того, як таблицю було заповнено, вона виводиться на консоль.

Екземпляри структури можна також створювати динамічним шляхом. Це реалізується як і для звичайних змінних, через виділення пам'яті. Але, відповідно до того, що екземпляр є ім'ям, яке пов'язано не з однією, а з декількома значеннями, використання його у програмі відрізняється від динамічних змінних.

Доступ до полів екземпляра здійснюється за допомогою оператора доступу через змінну-покажчик "->".

Розглянемо приклад програми, у якій динамічний масив з екземплярів структури задіяно для зберігання координат верхівок багатокутника.

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

struct point                          //оголошення структури точки на площині
{
    int x;                            //поле абсциси точки
    int y;                            //поле ординати точки
};

void main()                           //оголошення та визначення головної функції
{
    point *pP;                  //оголошення змінної-покажчика на структуру
    int num;                    //оголошення змінної визначення кількості точок
    printf("Enter number of points:"); //вивід запиту до користувача
    scanf("%i", &num);                //отримання значення від користувача
//динамічне створення масиву екземплярів структури
    pP=new point[num];
    for(int i=0; i<num; i++)          //цикл перебору елементів масиву
    {
        printf("Point No%i:\r\n", i);     //вивід номера точки
        printf("x: ");                //запит на ввід значення абсциси
        scanf("%i", &((pP+i)->x));  //отримання значення від користувача
        printf("y: ");                //запит на ввід значення ординати
        scanf("%i", &((pP+i)->y));//отримання значення від користувача
    }
    printf("You enter next values: \r\n"); //вивід константного рядка
    for(i=0; i<num; i++)              //цикл перебору елементів масиву
//вивід на консоль координат точок у круглих дужках
        printf("(%i, %i)\r\n", (pP+i)->x, (pP+i)->y);
}
```

На початку програми виконується підключення бібліотек для роботи з форматним вводом/виводом. Структура point включає два поля x та y для зазначення координат вершин багатокутника. У головній функції здійснюється оголошення змінної-покажчика на структуру, а потім для користувача виводиться запит про кількість точок, що буде записано у динамічний масив. Після створення масиву користувач повинен його заповнити. Коли всі координати було введено, їх значення виводяться на консоль.

Інші приклади використання структур буде розглянуто у темах "функції" та "пов'язані списки".

8.1.2. Поєднання

Ще одним складеним типом даних є поєднання. Воно не настільки поширене як структура, але також має своє призначення.

Поєднанням прийнято називати сукупність змінних, що займають у пам'яті одне й те саме місце.

Оголошення поєднання виконується через службове слово **union** за наступним форматом:

union <*ім'я поєднання*>
{
    <*оголошення членів*>;
} <*оголошення екземплярів поєднання* >;

Для того, щоб зрозуміти сутність поєднань, розглянемо схему (рис. 8.1), що відображає розміщені у пам'яті поля символьного, цілого та дійсного типів даних.

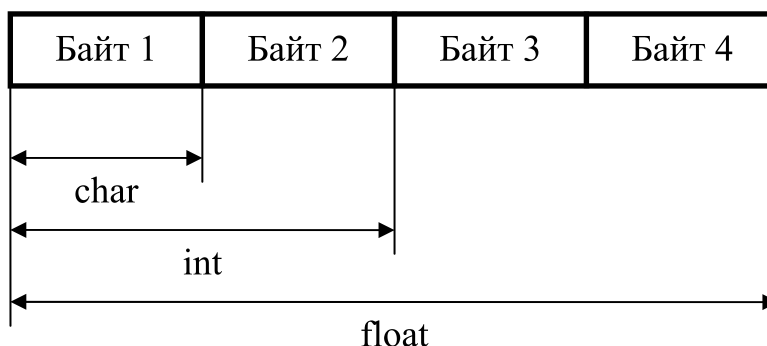

Рис. 8.1. Схематичне відображення розташування у пам'яті полів поєднання

Таким чином, при оголошенні трьох членів, що відповідають зазначеним вище типам даних, екземпляр поєднання буде займати у пам'яті чотири байти. Ця цифра збігається з кількістю байтів пам'яті, що виділяється на найбільший тип даних серед оголошених членів поєднань.

Основним призначенням для поєднань є групування та розбір даних. Так, наприклад, ціле значення може бути розкладено на два байти, завдячуючи наступному поєднанню:

```
union value                    //оголошення іменованого поєднання
{
    int V;                     //оголошення цілого поля
    char VB[2];                //оголошення поля символьного масиву
};
```

Як і для структур, екземпляри поєднань можна оголошувати у списку чи через ім'я типу даних користувача у програмі. Так, для наведеного вище прикладу поєднання у програмі оголошення екземпляру val може бути реалізовано, як вказано нижче:

```
value val;                     //оголошення екземпляру поєднання
```

Доступ до членів поєднань організовується так само, як і для структур: статичний екземпляр – "."; динамічний екземпляр – "->".

У мові C++ є окремий вид поєднань – безіменні. Їх призначення те саме, але екземплярів в них не буває. Доступ до членів реалізується безпосередньо через унікальні імена. Так, наприклад, у програмі можна використовувати наступний код:

```
union                          //оголошення безіменного поєднання
{
    float f;                   //оголошення поля з крапкою, що плаває
    int i;                     //оголошення цілого поля
};
f=7.15;                        //визначення поля поєднання
```

8.1.3. Перелічення

Перелічення – це складений тип даних, що є сукупністю іменованих цілих констант, що задають значення для змінної. Кожний член має ціле значення, що відповідає його порядковому номеру при оголошенні перелічення.

Оголошення виконується за допомогою службового слова **enum** за наступним форматом:

enum *<ім'я перелічення>* {*<список значень>*};

Наприклад, у програмі для роботи з назвами латинських літер оголошено наступне перелічення:

```
enum latin {alpha, beta, gama};          //оголошення перелічення
```

Відповідно до нього можна створити змінні:

```
latin l1, l2;                         //оголошення змінних перелічення
```

що визначатимуться:

```
l1=alpha;                             //визначення змінної
l2=gama;                              //визначення змінної
```

Для перелічень припустимо виконувати операції додавання та віднімання, але їх зміст не є інформативним. Так, наприклад, наступний код

```
latin l;                              //оголошення змінної перелічення

l=l2-l1;              //визначення змінної результатом віднімання
```

запише у l значення alpha.

Перелічення не мають явних призначень, як структури та поєднання. Їх застосування залишається на розсуд програміста.

Висновки

У даному розділи було розглянуто наступні основні питання:

– структури;
– перелічення;
– поєднання.

Контрольні питання

1) Що таке складений тип даних?
2) Яке поєднання прийнято називати структурою?
3) Наведіть формат оголошення структури.
4) Дайте визначення поняттю поле структури.
5) Чим для структури є екземпляр?
6) Як виконується робота з членами структури через екземпляр?
7) Для чого слугує список екземплярів структури при її оголошенні?
8) Скільки байтів у пам'яті займає екземпляр структури, що включає два цілочисельних та одне символьне поле?
9) Як створити динамічний екземпляр структури?
10) Як отримати доступ до членів структури через змінну-покажчик на екземпляр?
11) Що у мові C++ називають поєднаннями?
12) Чим поєднання відрізняються від структур?
13) Для чого слугують поєднання?
14) Як виконується робота з полями поєднань?
15) У чому суть застосування безіменних поєднань?
16) Наведіть формати оголошення іменованого та безіменного поєднань.

17) Дайте визначення поняттю перелічення.

18) Чим перелічення відрізняються від інших складених типів даних?

19) Як застосовуються перелічення?

20) Які операції можна виконувати зі змінними складеного типу даних перелічення?

9. ФУНКЦІЇ

Навчальною метою розділу є ознайомлення читача

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– оголошувати, визначати та викликати функції;

– виділяти закінчений програмний код у функцію;

– застосовувати змінні-покажчики та посилання у якості аргументів функцій;

– передавати у функцію масиви даних;

– задавати аргументи функції за умовчуванням;

– перевантажувати функції;

– застосовувати властивість рекурсії.

9.1. Процедурний підхід при написанні програм

Код будь-якої великої програми зазвичай містить безліч фрагментів, які або повністю, або з невеликими відмінностями повторюються два та більше разів. Такі фрагменти ускладнюють розуміння програми при її читанні. Запобігає цьому процедурний підхід мов програмування, що дозволяє об'єднувати програмні фрагменти в один блок, який може бути викликано декілька разів. У мові С++ такі програмні блоки називають функціями.

Функція – це програмний блок, що має унікальне ім'я, і включає групу виразів та операторів, що виконують одну або декілька логічно завершених дій. Усі вирази та оператори, що об'єднані у функцію, приховані від основної програми, – виділені в автономний програмний модуль, який іноді називають також підпрограмою. Взаємодія між основною програмою та такими модулями здійснюється через значення аргументів, що передаються та повертаються з функцій. Аргументи функцій є своєрідним інтерфейсом між її підпрограмою та програмістом.

Виділення логічно завершених програмних фрагментів у функції забезпечує можливість структуризації програми і зменшення програмного коду.

Для використання функцій в мові С++ існує три обов'язкових компоненти: оголошення, визначення та виклик.

9.1.1. Оголошення функції

Оголошення функції – це програмний запис, який дозволяє компілятору дізнатися про характеристики функції та виділити для існування її аргументів відповідний об'єм пам'яті. При оголошенні вказуються тип даних функції, її ім'я та список змінних, що будуть слугувати у якості аргументів. Формат оголошення функції наступний:

<*тип даних* ><*ім'я функції*>(<*список аргументів*>);

<*тип даних*> визначає тип значення, що повертає функція (окрім типу даних **void**). Для будь-якої функції за синтаксисом мови С++ існує лише єдине значення, що може бути повернуто. Якщо функція має тип даних **void**, то

значення, що повертається, відсутнє. <*ім'я функції*> у межах програми є унікальним ідентифікатором програмного модуля, що сформовано у функцію. Також як і ім'я змінної, ім'я функції не може містити розділових знаків і починатися з цифр. Найчастіше для іменування функцій використовують одне чи декілька повних або скорочених слів, що характеризують її призначення. Якщо таких слів декілька, то зручно кожне з них починати з великої літери. Це дозволяє спростити розуміння назв функцій. <*список аргументів*> є списком оголошень змінних, що передаються у функцію.

При оголошенні функції можна використовувати повний чи скорочений варіант списку аргументів, який не містить імен змінних. Якщо немає необхідності в передачі значень, то список аргументів може бути відсутнім.

Розглянемо приклади оголошення функцій з трьома аргументами, що повертає ціле значення, з повним та скороченим списком аргументів:

```
//оголошення функції  з повним списком аргументів
int func(int n, char c, float f);

//оголошення функції  зі скороченим списком аргументів
int func(int, char, float);
```

9.1.2. Визначення функції

Визначення функції включає відповідний програмний модуль, що слугує у якості підпрограми. Визначення включає рядок, відповідний до оголошення і логічно завершений програмний фрагмент, який називають тілом функції. Формат визначення функції наступний:

```
<тип даних><ім'я функції>(<список аргументів>)
{
	<вирази та оператори>
	return <значення, що повертається>;
}
```

Значення, що потрапляють до функції через аргументи використовуються у виразах та операторах тіла функції для вирішення задачі призначення модуля. Коли для функції було зазначено параметр, що буде повернуто, то для реалізації цього застосовується службове слово **return.**

return – це оператор, який визначає, що результат наступного за ним виразу буде передано в точку виклику функції як значення, що повертається. Якщо для функції було задано тип даних **void**, то **return** і наступний за ним вираз не потрібні.

9.1.3. Виклик функції

Виклик функції – це запис у програмному коді, який визначає місце вставки програмного модуля та задає значення аргументам функції. При

наявності параметра, що повертається, у точку виклику функції вставляється отримане значення. Формат виклику функції наведено нижче:

<*ім'я функції*>(<*список значень, що передаються*>);

Виклик функції може бути присвоєно змінній з відповідним типом даних для збереження повернутого значення. Також виклик функції може використовуватись у якості будь-якого виразу.

Функцію не можна викликати без її попередньої оголошення і визначення, як і не можна використовувати неоголошену і невизначену змінну у виразі. Проте, якщо для змінної оголошення та визначення можуть існувати як два окремих записи, то для функції визначення завжди включає оголошення. Таким чином, перед викликом функції достатньо виконати лише її визначення. Але наявність для функції оголошення надає програмісту можливість поліпшення структуризації програми. При використанні лише визначень виникає необхідність замислюватись про порядок їх розташування, а при великій кількості функцій і зовсім втрачається можливість організовувати правильну черговість для коректної роботи.

При оголошенні функції компілятор не розглядає її код, а, отже, немає значення місце її розташування серед інших оголошень. Вона лише повідомляє компілятору, що десь в межах програми обов'язково буде присутнє її визначення. Таким чином, за допомогою оголошень функцій розміщення програмних модулів стає довільним і залишається на розсуд програміста.

Розглянемо приклад програми, у якій застосовано функцію обчислення факторіала вказаного користувачем числа.

```
#include <stdio.h>                //підключення бібліотеки вводу/виводу

int factorial(int n);         //оголошення функції розрахунку факторіала числа

void main()                       //оголошення та визначення головної функції
{
        printf("Enter integer value:");   //вивід на консоль запиту до користувача
        int n; //оголошення змінної для отримання значення, що ввів користувач
        scanf("%i", &n);              //отримання значення з консолі
        if(n>0)                       //якщо факторіал може бути розраховано
//вивід на консоль результату виклику функції розрахунку факторіала числа
                printf("Factorial of %i is: %i", n, factorial(n));
        else                          //якщо факторіал не може бути розраховано
                printf("Wrong value");    //вивід на консоль статичного рядка
}

int factorial(int n)          //визначення функції розрахунку факторіала числа
{
```

```
//оголошення та визначення змінної для обчислення факторіала
	int f=1;
	if (n>1)			//якщо значення більше за одиницю
//цикл перебору цілих значень від 2 до n
		for (int i=2; i<=n; i++)
			f*=i;  //розрахунок факторіалу  для поточного значення  i
	return f;			//повернення отриманого значення факторіала
}
```

Роботу з консоллю засновано на використанні функцій форматного вводу та виводу, тому на початку програми підключається бібліотека stdio.h*. Перед точкою входу у програму виконується оголошення функції для розрахунку факторіалу. Визначення її йде за* main()*. У головній функції користувачу виводиться запит на ввід значення для розрахунку факторіалу і, якщо було введено число більше за* 0*, то реалізується виклик функції* factorial()*. Функція розрахунку факторіалу має один вхідний та один вихідний параметр. Розрахунок виконуються шляхом покрокового множення у циклі з переприсвоюванням значення змінної.*

У наведеному прикладі двічі оголошено змінну n, проте це не є помилкою – змінні знаходяться у різних програмних блоках, а, отже, не взаємодіють і не перетинаються в пам'яті. Під час виклику функції factorial() значення, що міститься у змінній n основної функції, переприсвоюється аргументові n і далі використовується у тілі функції. Які б дії не відбувалися з аргументом, це ніяк не відбивається на значеннях змінних, що існують у коді, з якого було виконано виклик. Виключенням є лише використання у якості аргументів посилань і змінних-покажчиків.

В якості аргументів у функцію можна передавати змінні, константи, екземпляри структури чи класу, посилання та змінні-покажчики.

9.1.4. Екземпляри структур у якості аргументів функції

Розглянемо приклад, де застосовано функцію, до якої передається екземпляр структури, що має два поля для зберігання цілих значень

```
#include <stdio.h>		//підключення бібліотеки вводу/виводу

struct znach			//оголошення структури для зберігання двох значень
{
	int x;		//оголошення поля  x
	int y;		//оголошення поля y
};
int sum (znach z);//оголошення функції розрахунку суми значень полів структури

void main()		//оголошення та визначення головної функції
{
```

```
	znach s;	//оголошення екземпляра структури znach
	s.x = 1;	//визначення поля x
	s.y = 8;	//визначення поля y
//вивід на консоль суми значень полів
	printf("%i + %i = %i", s.x, s.y, sum(s));
}
// визначення функції розрахунку суми значень полів структури
int sum (znach z)
{
	return z.x+z.y;	//отримання та повернення результату розрахунку
}
```

Для роботи з консоллю у програмі підключено бібліотеку форматного вводу/виводу. Потім виконується оголошення структури з двома полями. За нею йде оголошення функції, що буде використано для розрахунку суми значень полів. Робота з екземпляром структури у головній функції ведеться як і зі звичайними змінними – визначення полів і передача у якості аргументу. Результат розрахунку повертається до основного коду і виводиться на консоль.

Екземпляри структури також можуть бути застосовані для отримання декількох значень у якості результату виконання функції. Для цього тип даних функції треба зазначити у відповідності з назвою структури.

9.1.5. Посилання та змінні-покажчики у якості аргументів функції

При застосуванні посилань та змінних-покажчиків, значення змінних основної програми можуть бути змінені.

Механізм посилань – це особливість мови C++, що дозволяє передавати у функцію значення змінної через її псевдоім'я, яке пов'язано з адресою змінної у пам'яті.

Посилання та змінні-покажчики створюють можливість отримувати з функції за допомогою аргументів більш одного значення.

Особливістю посилань є те, що вони можуть бути задіяні лише у функціях. Спроба окремого оголошення посилання призведе до помилки. Для оголошення посилання використовується символ амперсанда – "&", який розміщується після типу даних аргументу і означає, що дана змінна буде виступати у якості псевдоім'я тієї, котра зазначена при виклику. Передати константне значення за посиланням неможливо.

Розглянемо приклад, у якому для впливу на значення змінних головного коду задіяно дві функції, з аргументом посиланням та аргументом змінною-покажчиком.

```
#include <stdio.h>			//підключення бібліотеки вводу/виводу

void ChangeValue1(int& n);	//оголошення функції з аргументом-посиланням
```

```
//оголошення функції з аргументом змінною-покажчиком
void ChangeValue2(int *n);

void main()                    // оголошення та визначення головної функції
{
	int k = 7;                 //оголошення і визначення змінної  k
	ChangeValue1(k);           //змінення значення k
	int m = 9;                 //оголошення і визначення змінної  m
	ChangeValue2(&m);          //змінення значення m
	printf("k = %i; m = %i", k, m); //вивід значень змінних на консоль
}

//визначення функції, що змінює значення змінної за посиланням
void ChangeValue1(int& n)
{
	n = 12;                //визначення змінної за посиланням
}

//визначення функції, що змінює значення змінної за змінною-покажчиком
void ChangeValue2(int *n)
{
	*n = 15;               //розіменування змінної-покажчика для визначення
}
```

Для роботи з консоллю задіяна бібліотека форматного вводу/виводу. Обидві функції, що використовуються для впливу на значення змінної, мають змістовно однаковий код. Під час виклику ChangeValue2() *для отримання адреси змінної m застосовано унарну операцію* "&".

Результатом виконання програмного коду буде вивід на консоль рядка:

```
k = 12; m = 15
```

Використання посилань та змінних-покажчиків у якості аргументів еквівалентні. Проте, якщо при передачі змінної-покажчика виникає необхідність перевірки її значення для запобігання збою програми за причиною невизначеності, то для посилання це виконувати немає необхідності, бо через нього можна передавати у функцію лише існуючі змінні.

Але на рівні явного недоліку змінних-покажчиків при їх застосуванні у якості аргументів, у порівнянні з посиланнями, є і перевага, бо лише так можна передавати у функцію масиви.

Розглянемо приклад програми, де функція використовується для отримання найменшого та найбільшого значень заданого масиву.

```
#include <stdio.h>                //підключення бібліотеки вводу/виводу
```

```
struct MaxMin            //оголошення структури для зберігання цілих значень
{
        int max;         //оголошення поля максимального значення
        int min;         //оголошення поля мінімального значення
};
//оголошення функції пошуку максимального та мінімального значень масиву
MaxMin find(int *pArr, int n);

void main()              //оголошення та визначення головної функції
{
//оголошення та ініціалізація масиву з 10 елементів
        int Arr[10] = {3,12,8,31,0,-3,2,-17,6,22};
//виклик функції пошуку значень для визначення полів екземпляра структури
        MaxMin m= find(Arr,10);
//вивід отриманих значень на консоль
        printf("Max = %i; Min = %i", m.max, m.min);
}

//визначення функції пошуку максимального та мінімального значень масиву
MaxMin find(int *pArr, int n)
{
        MaxMin m;              //оголошення екземпляра структури
        m.max = *pArr;   //визначення початкового значення поля структури
        m.min = *pArr;   //визначення початкового значення поля структури
        for(int i = 0 ; i < n; i++) //цикл перебору елементів масиву
        {
//якщо значення поточного елемента більше за max
                if(*(pArr+i)>m.max)
                        m.max = *(pArr+i);//записати у поле max нове значення
// якщо значення поточного елемента менше за min
                if (*(pArr+i)<m.min)
                        m.min = *(pArr+i); //записати у поле min нове значення
        }
        return m;   //повернення екземпляра структури
}
```

Робота з консоллю реалізується за допомогою форматних функцій, що визначені у підключеній бібліотеці stdio.h. *Структура* MaxMin *слугує для збереження максимального та мінамального значень масиву, що статично оголошено у головній функції. Для визначення полів структури застосовано виклик функції* find()*, до якої у якості аргументів передаються змінна-покажчик на масив і кількість елементів масиву.*

Робота з масивом, який передано у функцію, відповідає використанню динамічних масивів, тому для запобігання помилок доступу до неіснуючих елементів необхідно використовувати значення аргументу, що містить кількість елементів. Таким чином, для передачі масиву у функцію застосовується два аргументи.

9.1.6. Визначення аргументів за умовчуванням

Значення аргументів необов'язково повинні визначатися при виклику функції. У мові С++ існує механізм визначення аргументів за умовчуванням. Тобто, навіть якщо функція має список аргументів, то при її виклику дужки можна залишати порожніми. Визначення аргументів за умовчуванням виконується при оголошенні функції за наступним форматом:

<тип даних> <ім'я функції> (*<оголошення аргументу 1>* = *<значення 1>*,
<оголошення аргументу 2> = *<значення 2>*,…
<оголошення аргументу N> = *<значення N >*);

Як вже було сказано вище, оголошення аргументів можна виконувати без вказування імен змінних, таким чином, оголошення функції із значеннями аргументів за умовчуванням може мати два вигляди:

```
int func(int n = 15, char c = 'a', float f = 5.12);
```

або

```
int func(int = 15, char = 'a', float = 5.12);
```

При виклику такої функції в програмі немає необхідності визначати її параметри. Якщо залишити дужки порожніми, то для визначення будуть використані значення, що були вказані при оголошенні. Проте, не завжди значення за умовчуванням відповідають тим, що необхідно передавати у функції в коді програми. Визначення аргументів за умовчуванням не створює перешкод для їх перевизначення при виклику. Компілятор спочатку перевіряє, чи були передані значення при виклику, і лише потім звертається до значень за умовчуванням.

Під час виклику функцій зі значеннями параметрів за умовчуванням можна визначати не всі аргументи, а лише певну їх кількість з початку списку. Наприклад, для наведеної вище функції, її виклик може містити змінну кількість значень для аргументів, – від жодного до трьох:

```
func();           //всі аргументи визначено за умовчуванням

func(10);         //два останніх аргументи визначено за умовчуванням

func(10,'b');     //останній аргумент визначено за умовчуванням
```

```
func(10,'b',7.3);     //жоден аргумент не визначено за умовчуванням
```

Всі виклики функції func() можуть мати місце в програмі при наявності визначень її аргументів за умовчуванням при оголошенні. Проте, слід пам'ятати, що необхідність змінити значення лише для 3-го аргументу спричинить примусову зміну значень 1-го і 2-го. Тому при оголошенні функцій із значеннями параметрів за умовчуванням список аргументів упорядковується за пріоритетом можливої зміни значень при виклику.

9.2. Клас пам'яті змінних функції

Коли говорять про функції, то часто згадують поняття області видимості і класу пам'яті для змінних. Як вже було зазначено раніше (див. розділ "Структура програми на мові C++"), кожна змінна згідно з місцем її оголошення може мати локальну чи глобальну область видимості.

Стосовно до функцій, локальна область видимості – це використання змінної лише у тілі підпрограми, а глобальна – у поточному файлі. При локальній області видимості змінною не можна скористатися з інших функцій, бо час її існування обмежено лише програмним модулем, де її було оголошено. Якщо необхідно створити глобальну змінну, то її оголошення виконується поза всіма функціями перед першим її використанням.

Такі особливості визначає клас пам'яті змінної. Локальні змінні мають автоматичний клас пам'яті (automatic), а глобальні – статичний (static). Автоматичний клас пам'яті повідомляє компілятору, що для змінної потрібно виділити пам'ять лише до кінця виконання функції, а статичний – до кінця існування програми. Проте, хоч для локальних і глобальних змінних клас пам'яті визначається автоматично, для локальних змінних його можна змінити. Так, якщо перед типом даних при оголошенні локальної змінної написати службове слово **static**, то при завершенні роботи функції область пам'яті, що виділена для змінної, не буде вивільнена і проіснує до кінця виконання програми, причому під час кожного нового виклику функції з'явиться можливість використовувати останнє значення змінної.

Розглянемо приклад програми, де для подальшого використання значення локальної змінної функції було змінено клас пам'яті.

```
#include <stdio.h>          //підключення бібліотеки вводу/виводу
int sum();                  //оголошення функції розрахунку суми
int n = 5;                  //оголошення і визначення глобальної змінної  n

void main()                 //оголошення і визначення головної функції
{
//вивід на консоль результату виконання функції
	printf("%i\r\n", sum());
//вивід на консоль результату виконання функції
	printf("%i\r\n", sum());
```

```
// вивід на консоль результату виконання функції
	printf("%i\r\n", sum());
}

//визначення функції розрахунку суми
int sum()
{
	static int m = 0;  //оголошення і визначення статичної локальної змінної  m
	m++;          //інкрементування змінної  m
	n++;          //інкрементування змінної  n
	return m+n;         //повернення суми значень змінних
}
```

Для роботи з консоллю підключено бібліотеку форматного вводу/виводу. У головній функції три рази виконується виклик функції для розрахунку суми статичної локальної та глобальної змінних. Результат виконання виводиться на консоль. Кожного разу під час виклику у тілі функції обидві змінні інкрементуються. Результат виконання програми буде наступним:

```
7
9
11
```

Таким чином, зміна класу пам'яті для локальних змінних функції може бути використана для того, щоб отримане значення зберігалося між викликами і слугувало для подальших розрахунків у тілі тієї самої підпрограми.

9.3. Перевантаження функцій

Однією з особливостей застосування функцій у мові С++ є її перевантаження.

Перевантаження функції – це процес, при якому у межах однієї програми може існувати декілька функцій з одним ім'ям, що відрізняються між собою типом даних або числом аргументів.

Наприклад, для вже розглянутого прикладу програми, де застосовано функцію, у яку передається екземпляр структури, до коду може бути додано ще одну реалізацію функції розрахунку суми:

```
#include <stdio.h>                 //підключення бібліотеки вводу/виводу

int sum(int, int);                 //оголошення першої функції sum
struct znach                //оголошення структури для зберігання цілих значень
{
	int x;                    //оголошення змінної  x
	int y;                    //оголошення змінної  y
};
```

```
int sum(znach);                 //оголошення другої функції sum

void main()                     //оголошення та визначення головної функції
{
        int x = 7, y = 2;       //оголошення і визначення змінних
// вивід на консоль суми значень змінних
        printf("%i + %i  =  %i\r\n", x, y, sum(x, y));
        znach s;                //оголошення екземпляра структури
        s.x = 1;                //визначення поля x
        s.y = 8;                //визначення поля y
//вивід на консоль суми значень полів
        printf("%i + %i = %i", s.x, s.y, sum(s));
}

int sum(int x, int y)           // визначення першої  функції  sum
{
        return x+y;             // повернення результату розрахунку суми
}

int sum(znach z)                //визначення другої функції  sum
{
        return z.x+z.y;         //повернення результату розрахунку суми
}
```

У наведеному прикладі використані дві функції з ім'ям sum. Компілятор при обробці виклику перевіряє тип даних значень, що передаються, і/або їх кількість і співвідносить з ним необхідне визначення функції.

Перевантаження функцій вносить у програми на мові C++ універсальність, дозволяючи створювати бібліотеки, які включають функції з однаковими назвами для обробки різних типів значень, що мають розширені та спрощені форми.

9.4. Рекурсія функцій

Ще однією особливістю функцій є рекурсія. Так називають процес, при якому функція може у тілі викликати саму себе. Рекурсія дозволяє організувати складний програмний цикл. Проте користуватися цим механізмом необхідно з обережністю – кожен раз при виклику функції виділяється область пам'яті для її виконання, яка звільнюється лише після її завершення. Таким чином, скільки разів спрацьовує рекурсивний виклик, стільки блоків пам'яті виділяється під одну й ту саму функцію. При некоректному використанні рекурсії пам'ять, якою володіє процес, може бути витрачено повністю, що призведе до завершення програми.

Проте, хоча рекурсивний виклик і потребує обережного використання, бувають випадки, коли без нього практично неможливо обійтись. Так,

достатньо відомий метод швидкого сортування простіше всього може бути реалізовано тільки через рекурсію функції.

Наприклад, розглянутий на початку розділу приклад програми використання функції для знаходження факторіала числа, що вводить користувач, може бути виконано через рекурсію наступним чином:

```
#include <stdio.h>              //підключення бібліотеки вводу/виводу
int factorial(int);         //оголошення функції  розрахунку факторіала числа

void main()                     //оголошення та визначення головної функції
{
        printf("Enter integer value:");  //вивід запиту користувачу
//оголошення змінної для отримання введеного користувачем значення
        int n;
        scanf("%i",&n);         //отримання значення з консолі
        if(n>0)                 //якщо факторіал може бути розраховано
//вивід на консоль результату виклику функції розрахунку факторіала числа
                printf("Factorial of %i is: %i", n, factorial(n));
        else                    //якщо факторіал не може бути розраховано
                printf("Wrong value");  //вивід на консоль статичного рядку
}
int factorial(int n)        //визначення функції  розрахунку факторіала числа
{
        if (n>1)                //якщо значення більше за одиницю
//повернення значення розрахунку факторіала
                return n*factorial(n-1);
        return 1;               //повернення одиниці
}
```

Відмінність між програмами полягає у реалізації тіла функції factorial(). *У поточному прикладі застосовано рекурсію функції. Функції з використанням рекурсії значно менше, проте є більш складними для розуміння.*

Як вже було зазначено, рекурсія є своєрідним циклом, тому важливо при її застосуванні планувати умову для завершення. Виконання рекурсії проходить покроково, причому найчастіше для кожного нового виклику значення аргументів змінюються. Такий підхід дозволяє вирішувати завдання пошуку, видалення, сортирування тощо.

9.5. Застосування функцій

Таким чином, функції є повністю автономними програмними модулями, що призначені для вирішення підзадач. Вони слугують основною структурною одиницею при формуванні як процедурних, так і об'єктно-орієнтованих програм. Особливості застосування функцій роблять їх гнучким і корисним знаряддям при розробці як простих, так і складних програмних проектів.

Добре продумана та відлагоджена підпрограма може застосовуватись безліч разів для різного програмного забезпечення без внесення до неї будь яких змін. Окрім того, один проект може бути розроблено декількома програмістами, якщо його буде поділено структурно між виконавцями. Такими структурними елементами можуть слугувати бібліотеки функцій для вирішення поставлених підзадач.

Висновки

У даному розділи було розглянуто наступні основні питання:
– процедурний підхід при написанні програм;
– оголошення, визначення та виклик функцій;
– посилання та змінні-покажчики у якості аргументів функції;
– перевантаження функцій;
– рекурсія функцій.

Контрольні питання

1) Чому функції називають основними структурними одиницями процедурних мов?

2) Які три обов'язкових компоненти функцій потрібно зазначити у програмі для їх використання?

3) Яка різниця між викликом та визначенням функції?

4) Що називають списком аргументів функції?

5) Чому оголошення функції не є обов'язковим?

6) В чому полягає різниця між повним та скороченим списками аргументів?

7) Що можна передавати до функції у якості аргументів?

8) Як зазначити, що для функції не буде існувати параметра, який повертається?

9) Що прийнято називати посиланням?

10) Яке призначення має використання змінних-покажчиків у якості аргументів?

11) Які недоліки має застосування посилань у якості аргументів?

12) Як передати масив до функції?

13) Які підходи можна використовувати для отримання більше за одне значення з функції?

14) Що стає з локальною змінною при зміні для неї класу пам'яті?

15) Яке службове слово змінює клас пам'яті локальних змінних?

16) Що таке визначення аргументів за умовчуванням?

17) Якого правила необхідно дотримуватись при визначенні аргументів за умовчуванням?

18) У чому полягає сутність механізму перевантаження функцій?

19) Що називають рекурсією функції?

20) До чого може призвести використання великої кількості рекурсивних викликів?

10. ПОВ'ЯЗАНІ СПИСКИ

Навчальною метою розділу є ознайомлення читача з організацією та оперуванням пов'язаними списками.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– створювати однозв'язні та двозв'язні списки на базі екземплярів структур;

– додавати нові елементи до списку;

– вилучати елементи зі списку;

– працювати з FIFO та LIFO буферизацією;

– перебирати елементи списку за допомогою операторів циклу.

10.1. Пов'язані списки

Ще однією формою для роботи з великими об'ємами даних у мові C++ є пов'язані списки. Їх походження полягає з симбіозу між динамічними масивами та структурами. Слово "список" використовується невипадково, кожен елемент такого формування є екземпляром обраної в якості базової структури. Між собою елементи пов'язуються завдяки полям, що містять адреси екземплярів. Список створюється шляхом динамічного виділення пам'яті під екземпляр структури та подальшого його зв'язування з вже існуючими елементами. Особливістю такого формування є те, що при виконанні програми завжди задіяно стільки елементів списку, скільки потребує вирішення завдання. Створення та видалення елементів реалізується динамічним шляхом.

Пов'язані списки за кількістю зв'язувань між елементами поділяють на однозв'язні, двозв'язні та багатозв'язні. Використання пов'язаних списків у програмах для вирішення завдань, що потребують роботи з великими об'ємами даних однакового типу чи однакової належності, обмежено лише фантазією програміста.

Впорядкованість елементів списку у пам'яті не має потреби.

10.1.1 Однозв'язний список

Однозв'язні списки формуються у відповідності до схеми, наведеної на рис. 10.1.

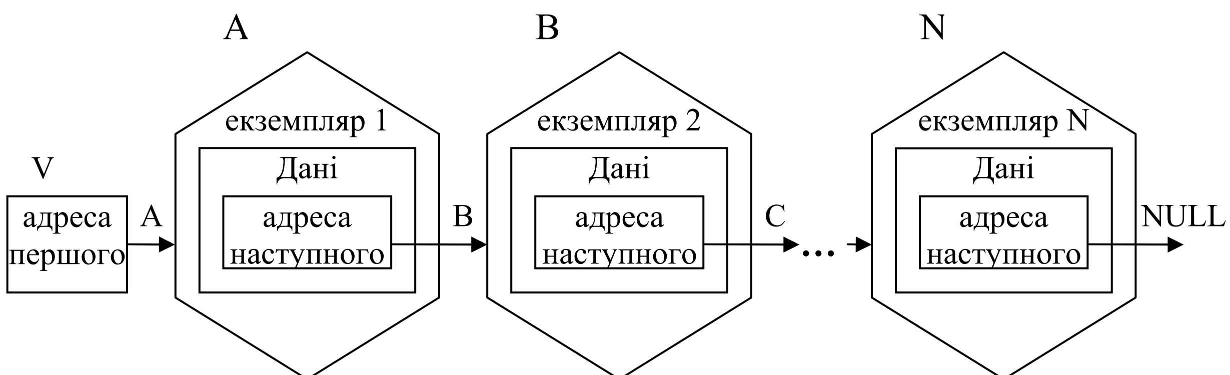

Рис. 10.1. Схематичне відображення однозв'язного списку

У зв'язку з тим, що список є повністю динамічним формуванням, для доступу до його елементів у програмі повинно бути створено змінну, що буде зберігати адресу одного з екземплярів, найчастіше першого або останнього. Іноді таких змінних може бути декілька.

В однозв'язному списку останній елемент у полі адреси зв'язку має значення NULL, що відповідає відсутності наступного екземпляра.

Створення нового елемента однозв'язного списку виконується згідно з наступним алгоритмом:

1) динамічне створення екземпляра базової структури
2) пошук попереднього елемента щодо того, котрий створюється у існуючому списку
3) ініціалізація поля адреси зв'язку
4) заповнення полів нового елемента

Черга виконання пунктів алгоритму не важлива, головне, щоб спочатку було створено новий екземпляр.

У залежності від місця, яке повинен посісти елемент, що створюється, етапи 2 та 3 можуть приймати різну форму.

Видалення існуючого елемента зі списку виконується за наступним алгоритмом:

1) пошук елемента, що повинен бути видалений
2) встановлення зв'язку між елементами списку в обхід елемента, що видаляється
3) вивільнення пам'яті, у якій було створено екземпляр

Розглянемо особливості роботи з однозв'язним списком на прикладі програми, що дозволяє користувачу ознайомитись з формуванням та обслуговуванням черги.

Чергою даних прийнято називати буфер зі значеннями, що обслуговується згідно з правилом FIFO (first in, first out – першим з'явився, першим обслуговується).

```
#include <stdio.h>              //підключення бібліотеки вводу/виводу

struct elem                     //оголошення структури для створення списку
{
        int value;              //оголошення поля для збереження значення
//оголошення поля для збереження адреси наступного елемента списку
        elem *pnext;
};

//оголошення глобальної змінної для адреси першого елемента списку
elem *pfirst;
void in_val(int);           //оголошення глобальної функції додавання значення
bool out_val(int&);         //оголошення глобальної функції вилучення значення
void show_q();              //оголошення глобальної функції виводу значень буферу
```

```
void main()                 //оголошення та визначення головної функції
{
	int val;                //оголошення змінної для вводу значення
	char ch;     //оголошення змінної для вводу значення виходу/продовження
	printf("Data Queue usage \r\n");        //вивід константного рядка
//вивід константного рядка
	printf("0-Quit, 1-Add value, 2-Get value, 3-Show Queue \r\n");
	pfirst=NULL;        //визначення змінної-покажчика за умовчуванням
	for(;;)                 //вічний цикл
	{
		printf("Make your choice: ");   //вивід запиту до користувача
		scanf("%i", &ch); //отримання значення від користувача
		if(ch==0)           //якщо користувач обрав вихід
			break;          //вихід з оператора
		else if(ch==1)      //якщо користувач обрав додавання значення
		{
			printf("Enter new value: ");     //вивід запиту до користувача
			scanf("%i", &val);//отримання значення від користувача
			in_val(val);          //виклик функції для запису значення
		}
		else if(ch==2)      //якщо користувач обрав вилучення значення
		{
			if(out_val(val))     //якщо є значення, то вилучити перше
				printf("Value is: %i \r\n", val); //вивід значення
			else                 //якщо значення відсутні
//вивід константного рядка
				printf("No values in Queue \r\n");
		}
		else if(ch==3)      //якщо користувач обрав вивід значень
			show_q();   //виклик функції виводу значень буфера
		else                //якщо користувач обрав інше значення
//вивід константного рядка
			printf("Your choice is not valid \r\n");
	}
}

void in_val(int val)       //визначення глобальної функції додавання значення
{
	elem *pnew=new elem; //динамічне створення екземпляра структури
	if(pfirst==NULL)          //якщо створений елемент є першим
		pfirst=pnew;        //запам'ятати його адресу
	else                        //якщо елемент не перший
	{
		elem *pcur; //оголошення змінної-покажчика поточного елемента
```

```
//визначення змінної-покажчика адресою першого елемента
                 pcur=pfirst;
                while(pcur->pnext!=NULL)    //цикл перебору елементів списку
                        pcur=pcur->pnext;  //перевизначення поточного елемента
                pcur->pnext=pnew; //пов'язати новий елемент з останнім у списку
        }
        pnew->value=val;                  //визначення значення елемента
        pnew->pnext=NULL;                 //визначення змінної зв'язку
}

bool out_val(int& val)   //визначення глобальної функції вилучення значення
{
        if(pfirst!=NULL)                  //якщо у списку є елементи
        {
                val=pfirst->value;    //отримати значення останнього елемента
//оголошення змінної-покажчика для зберігання адреси першого елемента
                elem *ptemp=pfirst;
                pfirst=pfirst->pnext;  //перевизначення адреси першого елемента
                delete ptemp;             //вивільнення пам'яті елемента
                return 1;                 //повернути 1
        }
        else                              //якщо у списку елементи відсутні
                return 0;                 //повернути 0
}
void show_q()              //визначення глобальної функції виводу значень буфера
{
        if(pfirst!=NULL)                  //якщо у списку є елементи
        {
                elem *pcur; //оголошення змінної-покажчика поточного елемента
//визначення змінної-покажчика адресою першого а
                pcur=pfirst;
                do                    //цикл перебору елементів списку
                {
//вивід значення поточного елемента
                        printf("%i   ", pcur->value);
                        pcur=pcur->pnext;  //перевизначення поточного елемента
                }
                while(pcur!=NULL);     //доки є елементи
                printf("\r\n");           //перехід на новий рядок
        }
        else
                printf("No values in Queue \r\n"); //вивід константного рядка
}
```

Програма дозволяє користувачу обирати одну з чотирьох можливостей: завершення, додавання нового значення до черги, отримання значення черги та вивід на консоль всіх значень черги. Останні три дії реалізовано через виклик відповідних до підзавдань функцій. Програма виконується у вічному циклі доки користувачем не буде обрано варіант завершення. Черга базується на структурі elem*, що має два поля:* value *– для зберігання значення елементу, та* pnext *– адресу наступного за чергою елемента. Функція* in_val() *у якості параметру приймає значення, що необхідно додати у чергу. Перший її рядок реалізує динамічне створення нового екземпляра структури. Потім виконується перевірка значення глобальної змінної* pfirst*, що зберігає адресу першого елемента списку, і у залежності від нього створений елемент додається як перший чи останній. Поля елемента заповнюються наприкінці функції. Функція* out_val() *через параметр посилання повертає значення першого елемента списку. При відсутності елементів параметр, що повертається приймає значення* 0*, в іншому випадку –* 1. out_val() *записує значення першого елементу у посилання, зберігає у* pfirst *адресу зв'язку і вивільнює пам'ять, що задіяна під перший елемент, видаляє зі списку. Функція* show_q() *при наявності елементів у списку перебирає їх з першого до останнього, поступово виводячи на консоль значення, що в них збережені. Захист від похибки дозволяє запобігти вводу користувачем невідповідного значення вибору та отримання значення з порожнього списку.*

Треба зазначити, що у вищенаведеній програмі зручніше було б, окрім змінної pfirst, також використовувати змінну plast, щоб не виникало необхідності при додаванні значень у чергу кожного разу виконувати перебір всього списку у пошуках останнього елемента. Застосований у прикладі підхід надає читачеві розуміння реалізації перебору елементів списку за допомогою циклів **while** та **do-while**.

10.1.2. Двозв'язний список

Також при програмуванні дуже поширеним є використання двозв'язного списку. Такі списки формуються за схемою, що наведено на рис. 10.2:

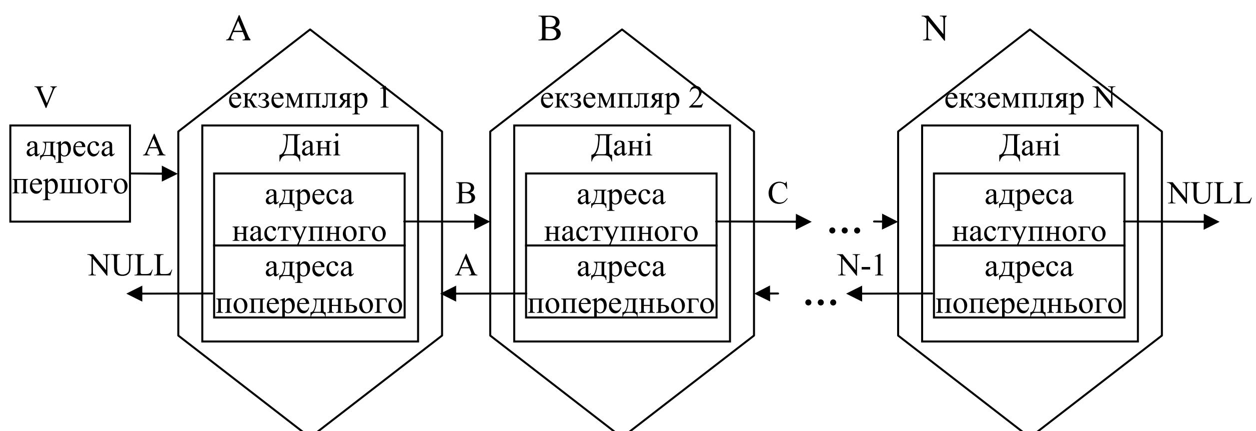

Рис. 10.2. Схематичне відображення двозв'язного списку

Як і для однозв'язного для доступу до двозв'язного списку з програми повина бути створена змінна. У двозв'язному списку поле адреси попереднього для першого та поле адреси наступного для останнього елементів повинні мати значення NULL.

Створення та видалення елементів виконується відповідно до однозв'язного списку, за виключенням того, що треба організовувати заразом два зв'язки.

Розглянемо особливості роботи з двозв'язним списком на прикладі програми, що знайомить користувача з формуванням та обслуговуванням стека.

Стеком даних прийнято називати буфер зі значеннями, що обслуговується за правилом LIFO (last in, first out – останнім з'явився, першим обслуговується).

```
#include <stdio.h>                    //підключення бібліотеки вводу/виводу

struct elem                   //оголошення структури для створення списку
{
        int value;            //оголошення поля для збереження значення
//оголошення поля для збереження адреси попереднього елемента списку
        elem *pprev;
//оголошення поля для збереження адреси наступного елемента списку
        elem *pnext;
};

//оголошення глобальної змінної адреси останнього елемента списку
elem *pcur;
void in_val(int );        //оголошення глобальної функції додавання значення
bool out_val(int&);       //оголошення глобальної функції вилучення значення
void show_s();            //оголошення глобальної функції виводу значень буфера

void main()               //оголошення та визначення головної функції
{
        int val;          //оголошення змінної для вводу значення
        char ch;    //оголошення змінної для вводу значення виходу/продовження
        printf("Data Stack usage \r\n");        //вивід константного рядка
//вивід константного рядка
        printf("0-Quit, 1-Add value, 2-Get value, 3-Show Stack \r\n");
        pcur=NULL;        //визначення змінної-покажчика за умовчуванням
        for(;;)                       //вічний цикл
        {
                printf("Make your choice: ");   //вивід запиту до користувача
                scanf("%i", &ch);         //отримання значення від користувача
                if(ch==0)                 //якщо користувач обрав вихід
                        break;            //вихід з оператора
                else if(ch==1)      //якщо користувач обрав додавання значення
```

```
		{
			printf("Enter new value: "); //вивід запиту до користувача
			scanf("%i",&val); //отримання значення від користувача
			in_val(val);		//виклик функції для запису значення
		}
		else if(ch==2)	//якщо користувач обрав вилучення значення
		{
			if(out_val(val))	//якщо є значення, то вилучити перше
//вивід значення
				printf("Value is: %i \r\n", val);
			else			//якщо значення відсутні
//вивід константного рядка
				printf("No values in Stack \r\n");
		}
		else if(ch==3)		//якщо користувач обрав вивід значень
			show_s();		//виклик функції виводу значень буфера
		else				//якщо користувач обрав інше значення
//вивід константного рядка
			printf("Your choice is not valid \r\n");
	}
}

void in_val(int val)	//визначення глобальної функції додавання значення
{
	elem *pnew=new elem;	//динамічне створення екземпляра структури
	if(pcur==NULL)			//якщо у списку елементи відсутні
//визначення змінної зв'язку з попереднім елементом
		pnew->pprev= NULL;
	else					//якщо у списку є елементи
	{
		pcur->pnext=pnew;	//зв'язати поточний елемент з новим
		pnew->pprev=pcur;	//зв'язати новий елемент з поточним
	}
	pcur = pnew;			//перевизначити поточний елемент
//визначення змінної зв'язку з наступним елементом
	pcur->pnext=NULL;
	pcur->value=val;		//визначення значення елемента
}

bool out_val(int& val)	//визначення глобальної функції вилучення значення
{
	if(pcur!=NULL)			//якщо у списку є елементи
	{
		val=pcur->value;	//отримати значення останнього елемента
```

```
//розірвання зв'язку між попереднім та поточним
            pcur->pprev->pnext= NULL;
//оголошення змінної-покажчика зберігання адреси поточного елемента
            elem *ptemp=pcur;
            pcur=pcur->pprev;  //визначення поточним попереднього елемента
            delete ptemp;            //вивільнення пам'яті елемента
            return 1;                //повернути 1
      }
      else                           //якщо у списку елементів немає
            return 0;                //повернути 0
}

void show_s()             //визначення глобальної функції виводу значень буфера
{
      if(pcur!=NULL)    //якщо у списку є елементи
      {
            elem *pfirst;//оголошення змінної-покажчика першого елемента
            pfirst =pcur;         //визначення змінної-покажчика
            while(pfirst->pprev!= NULL)  //цикл пошуку першого елемента
                  pfirst=pfirst->pprev;  //перевизначення першого елемента
            do                    //цикл перебору всіх елементів з першого
            {
                  printf("%i   ",pfirst->value);     //вивід значення елемента
                  pfirst=pfirst->pnext;  //перевизначення поточного елемента
            }
            while(pfirst!=NULL);     //доки є елементи
            printf("\r\n");          //перехід на новий рядок
      }
      else                           //якщо у списку елементів немає
            printf("No values in Stack \r\n");       //вивід константного рядка
}
```

Програма за принципом дії нагадує попередній приклад за виключенням того, що всі елементи мають два поля адрес зв'язку. Окрім того, значення, що отримується при запиті, є останнім з тих, що було додано у стек.

Треба зазначити, що як і у попередньому прикладі, лістинг можна зменшити, ввівши у програму додаткову глобальну змінну pfirst, що буде зберігати значення адреси першого елемента списку. Це позбавить від необхідності використання циклу пошуку першого елемента у функції show_s().

10.2. Застосування пов'язаних списків

Пов'язані списки є складним, але дуже потужним інструментом при роботі з даними. Окрім однозв'язних та двозв'язних списків також можуть знаходити застосування і багатозв'язні списки.

Елементи списків необов'язково повинні мати однаковий базовий тип. Правильне застосування пов'язаних списків дозволяє досить широко їх застосовувати у програмному забезпеченні для моделювання, іграх, інтерпретаторах мов, графічних редакторах тощо.

Наприклад, для опису та роботи з графом станів може бути застосовано багатозв'язний список за схемою, що вказано на рис. 10.3.

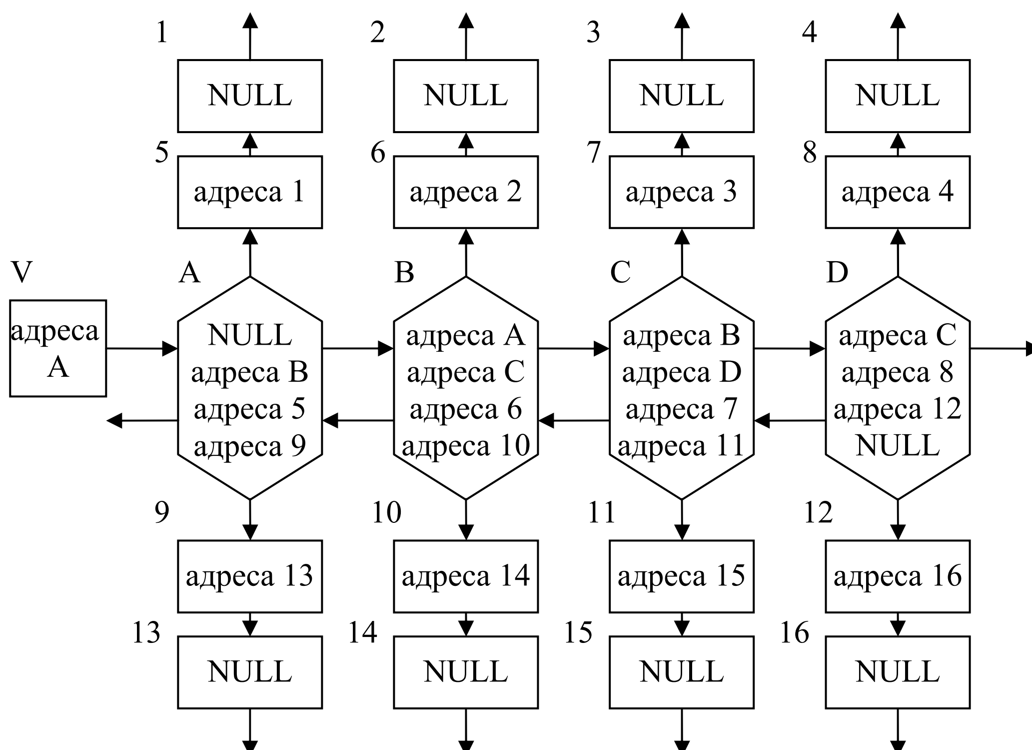

Рис. 10.3. Схематичне відображення багатозв'язного списку

Основою для створення пов'язаних списків може слугувати не тільки структура, але й клас. У мові С++ клас має багато спільного зі структурою.

Висновки

У даному розділи було розглянуто наступні основні питання:
– однозв'язний список;
– двозв'язний список.

Контрольні питання

1) Для чого слугують пов'язані списки?
2) Що прийнято називати пов'язаним списком?
3) Як виконується зв'язок між елементами у однозв'язному списку?
4) Через що з програмного коду може бути отримано доступ до пов'язаного списку?

5) За якою схемою створюється однозв'язний список?

6) За яким алгоритмом виконується створення нового елемента списку?

7) Як видалити елемент списку, що не є першим, або останнім?

8) Що таке FIFO буфер?

9) Що називають стеком?

10) Як видалити перший елемент однозв'язного списку, створеного за принципом FIFO?

11) Як за допомогою простого циклу з передумовою перебрати всі елементи однозв'язного списку?

12) За якою схемою створюється двозв'язний список?

13) Завдяки чому реалізовано зв'язок між елементами у двозв'язному списку?

14) За яким алгоритмом виконується видалення елемента двозв'язного списку?

15) Чому при роботі з пов'язаними списками з програми зручніше мати дві змінні-покажчика?

16) Як видалити останній елемент двозв'язного списку, створеного за принципом LIFO?

17) Як за допомогою простого циклу з постумовою перебрати всі елементи двозв'язного списку?

18) Чи має багатозв'язний список включати лише однотипні елементи?

19) Для вирішення якого класу задач може бути застосовано пов'язані списки?

20) На основі якого складеного типу даних створюються пов'язані списки?

ЧАСТИНА II
ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

11. ПОТОКОВИЙ ВВІД/ВИВІД ПРИ РОБОТІ З КОНСОЛЛЮ

Навчальною метою розділу є ознайомлення читача з потоковим вводом/виводом даних до консолі.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– застосовувати оператори **cin** та **cout** для організації вводу та виводу;

– форматувати інформацію за допомогою маніпуляторів формату;

– використовувати вивчені підходи реалізації вводу/виводу у відповідності до поставленного завдання.

11.1. Потоковий ввід/вивід

Окрім вже розглянутого форматного вводу/виводу, що було успадковано від C, мова C++ дозволяє використовувати потоковий ввід/вивід на основі операторів **cin** та **cout**.

Потоком вводу/виводу при роботі з програмним забезпеченням виступає логічний пристрій, що застосовується як інтерфейс для впорядкованої передачі даних.

Потоки вводу та виводу у C++ незалежні один від одного. Реалізація кожного з них виконується через свій оператор, згідно з чим іноді потік та оператор поєднують під одним поняттям: потік вводу **cin** та потік виводу cout.

Робота з потоками здійснюється через операції вилучення ">>" та вставки "<<", при чому вилучення застосовується лише до потоку вводу, а вставка – до виводу. Таким чином, через програмний код зазначається, що дані, що вводить користувач вилучаються з потоку вводу консолі, а дані з програми вставляються у потік виводу на консоль.

Використання потокового вводу/виводу при роботі зі змінними єпростішим у порівнянні з форматними, бо через потоки програма сама розуміє який тип даних вводиться або виводиться.

Формати потокових вводу та виводу наведено нижче:

cin >> *<змінна>* >> ...
cout << *<змінна>* <<...

Задавши потік одного разу, можна впродовж рядка скільки завгодно вилучати чи вставляти дані, чергуючи змінні з відповідними операціями.

Як і для форматного вводу/виводу у потоковому є можливість форматування тексту. Для цієї задачі використовуються маніпулятори, повний перелік яких наведено у табл. 11.1.

Таблиця 11.1

Маніпулятори

Маніпулятори	Призначення	Напрям
boolalpha	встановлює прапорець boolalpha	ввід-вивід

Закінчення табл. 11.1

Маніпулятори	Призначення	Напрям
dec	встановлює прапорець dec	ввід-вивід
endl	виводить символ переходу на новий рядок і очищує буфер	вивід
ends	виводить нульовий байт	вивід
fixed	встановлює прапорець fixed	вивід
flush	очищує буфер	вивід
hex	встановлює прапорець hex	ввід-вивід
internal	встановлює прапорець internal	вивід
left	встановлює прапорець left	вивід
noboolalpha	скидає прапорець boolalpha	ввід-вивід
noshowbase	скидає прапорець showbase	вивід
noshowpoint	скидає прапорець showpoint	вивід
noshowpos	скидає прапорець showpos	вивід
noskipws	скидає прапорець skipws	ввід
nounitbuf	скидає прапорець unitbuf	вивід
nouppercase	скидає прапорець uppercase	вивід
oct	встановлює прапорець oct	ввід-вивід
resetiosflags(fmtflags f)	скидає прапорці, вказані параметром f	ввід-вивід
right	встановлює прапорець right	вивід
scientific	встановлює прапорець scientific	вивід
setbase(int base)	задає основані системи числення, вказане параметром base	ввід-вивід
setfill(int ch)	задає символ-заповнювач ch	вивід
setiosflags(fmtflags f)	встановлює прапорці, вказані параметром f	ввід-вивід
setprecision(int p)	задає кількість цифр після десяткової точки	вивід
setw(int w)	задає ширину поля, вказану параметром w	вивід
showbase	встановлює прапорець showbase	вивід
showpoint	встановлює прапорець showpoint	вивід
showpos	встановлює прапорець showpos	вивід
skipws	встановлює прапорець skipws	ввід
unitbuf	встановлює прапорець unitbuf	вивід
uppercase	встановлює прапорець uppercase	вивід
ws	ігнорує провідні роздільники	ввід

Для того, щоб компілятор міг опрацьовувати код, який вміщує роботу з потоками, треба підключити наступні заголовні файли:

iostream.h – бібліотека, яка включає визначення операторів **cin** та **cout**

iomanip.h – бібліотека, що включає визначення маніпуляторів

Причому підключення можна виконувати без зазначення розширення ".h", якщо планується задіяти простір імен std.

Розглянемо використання потокового вводу/виводу на прикладі вже знайомої за вивченням форматного вводу/виводу програмою, що запитує у користувача значення і виводить їх на консоль.

```
#include <iostream>                 //підключення бібліотеки вводу/виводу
#include <iomanip>                  //підключення бібліотеки маніпуляторів

using namespace std;                //застосування простору імен

void main()                         //оголошення та визначення головної функції
{
        int i;                      //оголошення цілої змінної
        float f;                    //оголошення змінної з плаваючою точкою
        char c;
        cout <<"Enter int value:" <<flush;    //вивід запиту до користувача
        cin >>i;                    //отримання значення від користувача
        cout <<"int value is:" <<i <<endl;    //вивід значення на консоль
        cout <<"Enter float value:" <<flush;  //вивід запиту до користувача
        cin >>f;                    //отримання значення від користувача
//вивід значення та наступного запиту до користувача
        cout <<"float value is:" <<f <<endl<<"Enter char value:" <<flush;
        cin >>c;                    //отримання значення від користувача
        cout <<"char value is:" <<c <<endl;  //вивід значення на консоль
}
```

Для роботи з потоковим вводом/виводом та маніпуляторами підключено бібліотеки iostream *та* iomanip*. Необхідність задіяти простір імен* std *вказується за допомогою директиви* **using**. *Програма під час виконання почергово запитує значення для цілої, з точкою, що плаває, та символьної змінних, і після їх вводу виводить отримані дані на консоль.*

У наведеному прикладі показано, що маніпулятори, наразі **endl**, можуть бути використані як у середині, так і наприкінці рядка потоку.

Якщо провести аналогію між двома способами вводу/виводу, що було розглянуто, то потоковий є більш зручнішим при роботі з даними, але при форматуванні він значно програє. Це пов'язано з тим, що при використанні маніпуляторів у потоці зміни діють постійно на відміну від форматних функцій, де форматування виконується тільки над поточним рядком. Окрім того, для простих дій потрібно застосовувати більше одного маніпулятору. Але форматування за допомогою маніпуляторів має переваги, бо воно є більш гнучким.

Висновки

У даному розділи було розглянуто наступні основні питання:

– вивід на консоль за допомогою потокового оператора cout;

– ввід з консолі за допомогою потокового оператора cin;

– маніпулятори форматування.

Контрольні питання

1) Що називають потоком вводу/виводу?

2) За допомогою якого оператора реалізується потоковий вивід даних на консоль?

3) Наведіть формат потокового виводу даних.

4) Для чого застосовують маніпулятори?

5) Які операції використовують для роботи з потоками вводу/виводу?

6) Який заголовний файл містить визначення для роботи потокових операторів вводу/виводу?

7) За допомогою якого оператора реалізовано потоковий ввід даних з консолі?

8) У чому полягає різниця у застосуванні маніпуляторів **endl** та **flush**?

9) Який заголовний файл містить визначення для роботи з маніпуляторами?

10) Порівняйте форматний та потоковий ввід/вивід.

12. ОБ'ЄКТНО-ОРІЄНТОВАНИЙ ПІДХІД. КЛАСИ ТА ЇХ ЧЛЕНИ

Навчальною метою розділу є ознайомлення читача з об'єктно-орієнтованим підходом при розробці програмного забезпечення та класами.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– створювати класи;
– оголошувати та застосовувати об'єкти для роботи з членами класів;
– використовувати спеціальні методи класів;
– виконувати ініціалізацію полів класу за допомогою конструктора;
– реалізовувати інкапсуляцію;
– обмежувати доступ до членів класу за допомогою специфікаторів.

12.1. Об'єктно-орієнтоване програмування

В житті ми часто зустрічаємо поняття об'єкта. Так називають предмет, явище чи процес, що існує і має унікальні властивості, які його характеризують, та може бути задіяний тим чи іншим шляхом. Об'єкт може бути основою для проведення досліджень або слугувати метою для розробок.

З життя це поняття було запозичено і до програмування. Але програмний об'єкт не є існуючим насправді. Так прийнято називати сукупність даних та функцій для роботи з ними, що згідно із вказаною задачею характеризують об'єкт, що є реальним. Завдячуючи такій реалізації на базі програмного забезпечення, можна створювати моделі, що достатньо чітко повторюють процеси та явища. Окрім того, на відміну від процедурних мов програмування, об'єктно-орієнтований підхід надає можливість поєднання підпрограм та даних за призначенням у окремі модулі.

12.1.1. Клас

Основою при написанні об'єктно-орієнтованих програм є клас. Він реалізує функцію інкапсуляції, яка надає програмісту можливість використовувати структурні блоки коду через вбудований інтерфейс, не вдаючись у подробиці змісту їх побудови. Інкапсуляція є одним з трьох базових понять об'єктно-орієнтованого програмування.

Класом прийнято називати сукупність даних, поєднаних за загальним змістом, та функцій, що дозволяють застосовувати ці дані. Вони дуже схожі на структури. Ті й інші є складеними типами даних, можуть включати змінні стандартних типів даних та реалізовуватися через екземпляри та змінні-покажчики. Екземпляри класів при об'єктно-орієнтованому програмуванні часто називають об'єктами, що дозволяє відрізняти їх як окремі змінні при описі коду та зазначає їх призначення.

Створення класів виконується за допомогою службового слова **class** за наступним форматом:

class <*ім'я-класу*>
{
        < *оголошення функцій та змінних* >

<специфікатор_доступу 1>:
< оголошення функцій та змінних>
<специфікатор_доступу 2>:
< оголошення функцій та змінних>
.
.
.
<специфікатор_доступу N>:
< оголошення функцій та змінних>
}*<список об'єктів>*;

Список об'єктів як і для структур не є обов'язковим і застосовується лише тоді, коли є необхідність у створенні екземпляра глобальної видимості. Ім'я класу зазвичай обирається, зважаючи на відповідне його призначення. Кожне слово починається з великої літери і попереду дописується "C", що відповідає англійському class. Такий підхід дозволяє використовувати класи як програмні модулі, що можуть бути застосовані багатьма розробниками, як компоненти програм для вирішення великої кількості різноманітних завдань. Для цього оголошення класу виділяють у файл з розширенням ".h", а визначення всіх його методів у файл ".cpp" робить клас окремим структурним елементом (див. Багатофайлові проекти).

12.1.2. Доступ до членів класу

Функції та змінні, що включені до класу, є його членами, причому функції прийнято називати методами, а змінні, як і для структур, – полями. Часто говорять, що дані інкапсульовані у клас.

Доступ до полів та методів класу з основної програми може бути трьох видів: відкритий, закритий та захищений. Вид доступу визначається за допомогою спеціальних службових слів, що називаються специфікаторами доступу:

public – повний доступ (відкритий);

private – доступ лише з методів класу (закритий);

protected – можливість доступу з методів членів та з похідного класу (захищений).

Таким чином, робота з членами класу здійснюється за схемою, що наведено на рис. 12.1.

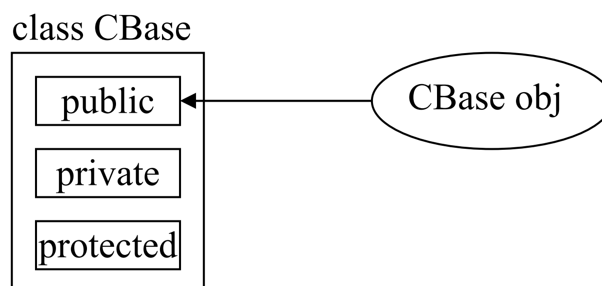

Рис. 12.1. Схематичне відображення доступу до членів класу

Якщо при оголошенні членів класу специфікатор доступу не було вказано, то за умовчуванням застосовується закритий доступ.

Визначення методів класу можна виконувати як у самому класі, так і за його межами. При цьому до імені функції, що визначається, зліва через операцію "::" додається ім'я її класу.

```
<тип даних > <ім'я класу>::<ім'я методу класу>(<аргументи функції>)
{
	<тіло функції>
}
```

Операцію "::" називають глобальним дозволом. Вона застосовується для того, щоб вказати належність функцій чи даних до відповідного класу.

Розглянемо приклад класу, що поєднує дані та функції роботи з відрізком на площині.

```
class CSegment            //оголошення класу для робот із відрізком
{
    public:               //відкритий доступ
        int x0;           //абсциса першої точки відрізка
        int y0;           //ордината першої точки відрізка
        int x1;           //абсциса другої точки відрізка
        int y1;           //ордината другої точки відрізка
//оголошення та визначення функції отримання координат точки центра
//відрізка
        void GetCenterPoint(int& x, int& y)
        {
            x=(x1+x0)/2;      //розрахунок абсциси точки центра
            y=(y1+y0)/2;      //розрахунок ординати точки центра
        }
//оголошення функції визначення довжини відрізка по осі абсцис
        int GetXLength();
//оголошення функції визначення довжини відрізка по осі ординат
        int GetYLength();
};
//визначення функції визначення довжини відрізка по осі абсцис
int CSegment::GetXLength()
{
    return (x0<x1)?(x1-x0):(x0-x1);
}
//визначення функції визначення довжини відрізка по осі ординат
int CSegment::GetYLength()
{
    return (y0<y1)?(y1-y0):(y0-y1);
}
```

В оголошеному класі членами є поля x0, x1, y0, y1 *для зберігання значень координат точок початку та кінця відрізка та методи* GetCenterPoint() – *отримання координати центра відрізка,* GetXLength() – *отримання довжини відрізка вздовж осі* X, GetYLength() – *отримання довжини відрізка вздовж осі* Y. *Оголошення методу* GetCenterPoint() *виконано у класі, а* GetXLength() *та* GetYLength() *за його межами.*

Застосування методів та полів класу у програмі відповідає роботі зі структурами. Спочатку потрібно оголосити об'єкт, а потім скористатися операцією доступу до членів – ".". Наприклад, використання розглянутого вище класу може полягати у наступному коді:

```
CSegment s;         //оголошення об'єкта класу
//визначеня координати першої точки
s.x0=10;
s.y0=25;
//визначеня координати другої точки
s.x1=20;
s.y1=49;
int x,y;            //оголошення змінних для отримання координати центра
//виклик методу для отримання координати центра відрізка
s.GetCenterPoint(x,y);
//вивід на консоль отриманої координати
cout<<"Center point: ("<<x<<","<<y<<")"<<endl;
//отримання і вивід на консоль довжини відрізка вздовж осі абсцис
cout<<"X Length: "<<s.GetXLength()<<endl;
//отримання і вивід на консоль довжини відрізка вздовж осі ординат
cout<<"Y Length: "<<s.GetYLength()<<endl;
```

12.1.3. Спеціальні методи класу

Два методи класу є особливими, бо не можуть існувати як окремі функції. Це конструктор та деструктор. Вони не мають типу даних і іменуються, як і сам клас. Різниця у їх назвах полягає лише у символі "~", що додається до назви деструктора.

Конструктор – це метод класу, який викликається автоматично при створенні об'єкта. Він використовується для визначення початкових значень полів, виділення пам'яті та захоплення ресурсів.

Деструктор теж викликається автоматично, але при знищенні об'єкта класу. У деструкторі виконуються дії, необхідні для коректного завершення роботи з даними та методами класу, – вивільнення пам'яті чи ресурсів.

Обидва методи існують у класах навіть, коли їх не було зазначено, бо становлять невід'ємну частину об'єктів. Але без явного оголошення та визначення конструктор та деструктор не можна використовувати для вирішення завдань програміста.

Особливістю використання конструктора є також список ініціалізації, що дописується після закриваючої круглої дужки списку аргументів та застосовується для визначення полів.

Автоматичний виклик спеціальних методів означає, що програміст не має можливості викликати їх де завгодно у програмному коді. Конструктор викликається в точці оголошення об'єкта і через нього може отримувати значення для своїх аргументів. На відміну від нього деструктор зовсім не має аргументів, бо можливість їх визначення відсутня. З автоматичним викликом також пов'язана відсутність у спеціальних методів значень, що повертаються. Їх не можливо було б отримати з програми. Крім того, конструктор і деструктор повинні при оголошенні бути включені до членів з відкритим доступом. Спроба застосувати інший специфікатор доступу призведе до помилки.

Як і для звичайних функцій, для методів, за виключенням деструктора, працює властивість перевантаження, що значно розширює можливості застосування класів. Перевантаження деструктора не може бути виконано за причиною відсутності в нього списку аргументів.

Розглянемо приклад програми, яка працює з класом, що дозволяє програмісту застосовувати пов'язаний з об'єктом динамічний масив.

```
#include <iostream>     //підключення бібліотеки потокового вводу/виводу
#include <iomanip>      //підключення бібліотеки маніпуляторів

using namespace std;    //застосування простору імен std

class CDynArr           //оголошення класу роботи з динамічним масивом
{
        int size;       //оголошення поля для зберігання розміру масиву
    public:             //відкриті члени класу
        int* pArr;      //змінна-покажчик для створення масиву
//оголошення конструктора класу без аргументів
        CDynArr() : size(10)            //список ініціалізації
        {
            pArr=new int[size];         //створення динамічного масиву
//вивід статичного рядка на консоль
            cout<<"Arrey with size 10 was created"<<endl;
        }
//оголошення конструктора класу з аргументом для визначення розміру масиву
        CDynArr(int sz) : size(sz)      //список ініціалізації
        {
            pArr=new int[size];         //створення динамічного масиву
//вивід статичного рядка на консоль
            cout<<"Arrey with size "<<sz<<" was created"<<endl;
        }
//оголошення методу запису значень у масив
        bool AddVal(int, int);
```

```
//оголошення методу читання значень елементів масиву
        bool GetVal(int, int&);
//оголошення методу отримання максимального значення масиву
        int GetMaxVal();
//оголошення методу отримання мінімального значення масиву
        int GetMinVal();
//оголошення методу отримання середнього арифметичного значення масиву
        float GetMidVal();
//оголошення деструктора класу
        ~CDynArr()
        {
                delete [] pArr;     //вивільнення пам'яті
//вивід статичного рядка на консоль
                cout<<"Arrey with size "<<size<<" was deleted"<<endl;
        }
};

//визначення методу запису значень у масив
bool CDynArr::AddVal(int pos, int val)
{
        if (pos > (size-1) || pos < 0)      //якщо вказаний елемент відсутній
                return 0;                   //повернути 0
        *(pArr+pos)=val;                    //записати значення до елемента
        return 1;                           //повернути 0
}

//визначення методу читання значень елементів масиву
bool CDynArr::GetVal(int pos, int& val)
{
        if(pos > (size-1) || pos < 0)       //якщо вказаний елемент відсутній
                return 0;                   //повернути 0
        val=*(pArr+pos);                    //читати значення елемента
        return 1;                           //повернути 0
}

//визначення методу отримання максимального значення масиву
int CDynArr::GetMaxVal()
{
//оголошення змінної для пошуку максимального значення
        int max=*pArr;              //визначення змінної першим значенням
        for(int i=1; i < size; i++) //цикл перебору всіх елементів з 2
                if(max < *(pArr+i))         //якщо значення більше за max
                        max=*(pArr+i);      //визначити max
        return max;                 //повернути отримане значення
}
```

```
//визначення методу отримання мінімального значення масиву
int CDynArr::GetMinVal()
{
//оголошення змінної для пошуку мінімального значення

	int min=*pArr;		//визначення змінної першим значенням
	for(int i=1; i < size; i++)//цикл перебору всіх елементів з 2
		if(min > *(pArr+i))	//якщо значення менше за min
			min=*(pArr+i);	//визначити min
	return min;			//повернути отримане значення
}

//визначення методу отримання середнього арифметичного значення масиву
float CDynArr::GetMidVal()
{
//оголошення змінної для розрахунку суми та її визначення
	float sum=0;
	for(int i=0; i < size; i++) //цикл перебору всіх елементів
		sum+=*(pArr+i);	//додавання поточного значення
	return sum/size;		//повернути середнє арифметичне
}

//оголошення та визначення головної функції програми
void main()
{
	int v;		//оголошення змінної для отримання значень користувача
	cout<<"Auto size:"<<endl;	//вивід на консоль статичного рядка
//оголошення змінної-покажчика на клас
	CDynArr *pda;		//оголошення змінної-покажчика на клас
	pda=new CDynArr;	//створення динамічного об'єкту
	int i=0;			//оголошення та визначення змінної лічильника
//цикл заповнення масиву об'єкта
	do
	{
//вивід на консоль рядка запиту
		cout<<"Enter "<<i<<" value: "<<flush;
//отримання значення від користувача
		cin>>v;
//визначення елемента масиву та перевірка умови циклу
	}while(pda->AddVal(i++, v));
//вивід на консоль знайденого максимального значення
	cout<<"Max value is: "<< pda->GetMaxVal()<<endl;
//вивід на консоль отриманого мінімального значення
	cout<<"Min value is: "<< pda->GetMinVal()<<endl;
//вивід на консоль отриманого середнього арифметичного значення
```

```
        cout<<"Mid value is: "<< pda->GetMidVal()<<endl;
        delete pda;          //вивільнення пам'яті

        cout<<"User size:"<<endl;     //вивід на консоль статичного рядка
//вивід на консоль запиту до користувача
        cout<<"Enter the arrey size (1...): "<<flush;
        cin>>v;                       //отримання кількості елементів масиву
        if(v > 0)                     //якщо значення більше за 0
        {
                i=0;                  //визначення змінної лічильника
//оголошення об'єкта та визначення аргументу конструктора
                CDynArr da(v);
//цикл заповнення масиву об'єкта
                do
                {
//вивід на консоль рядка запиту
                        cout<<"Enter "<<i<<" value: "<<flush;
//отримання значення від користувача
                        cin>>v;
//визначення елемента масиву та перевірка умови циклу
                }while(da.AddVal(i++, v));
//вивід на консоль знайденого максимального значення
                cout<<"Max value is: "<< da.GetMaxVal()<<endl;
//вивід на консоль отриманого мінімального значення
                cout<<"Min value is: "<< da.GetMinVal()<<endl;
//вивід на консоль отриманого середнього арифметичного значення
                cout<<"Mid value is: "<< da.GetMidVal()<<endl;
        }
        else                  //якщо значення менше чи дорівнює 0
//вивід на консоль статичного рядка
                cout<<"Wrong size!"<<endl;
}
```

Клас включає поля size *та* pArr *для створення динамічного масиву і методи* CDynArr() – *конструктори класу,* ~CDynArr() – *деструктор класу,* AddVal() – *запис значення у масив,* GetVal() – *читання значення з масиву,* GetMaxVal() – *отримання максимального значення серед елементів,* GetMinVal() – *отримання мінімального значення серед елементів,* GetMidVal() – *отримання середнього арифметичного для значень масиву. У залежності від того, було чи ні при оголошенні об'єкта вказано значення аргументу, виконується виклик першої чи другої реалізації конструктора. Список ініціалізації використовується задля визначення поля size. У тілах конструкторів виконується створення динамічного масиву. Деструктор вивільнює пам'ять, що була задіяна під масив. У головній функції робота з класом демонструється двома шляхами – через створення динамічного та статичного об'єктів. При створенні динамічного*

об'єкта виконується виклик конструктора без аргументів, а при створенні статичного – з аргументом.

Призначення класів є дуже широким, бо їх застосування не тільки виконує задачу функцій структуризації програми, але ще й робить код модульним, що застосовується при розробці великих проєктів, до яких задіяно більш одного програміста. Крім того, інкапсуляція та можливість створення окремих закритих бібліотек дозволяють використовувати класи як комерційний продукт для продажу.

Висновки

У даному розділи було розглянуто наступні основні питання:
– об'єктно-орієнтований підхід в програмуванні;
– класи та їх члени;
– спеціальні методи класів.

Контрольні питання

1) Дайте визначення поняттю клас.
2) Що називають інкапсуляцією?
3) У чому полягає призначення об'єктів класу?
4) Які методи класу є спеціальними?
5) Який метод зазвичай використовується для визначення даних класу за умовчуванням при створенні об'єкта?
6) Який метод призначено для вивільнення ресурсів при знищенні об'єкта?
7) Для чого використовується список ініціалізації?
8) Яке призначення має перевантаження конструктора?
9) Чому спеціальні методи не мають значень, що повертаються?
10) Чому деструктор не може бути перевантажено?
11) Для яких задач може знадобитися перевантаження конструктора?
12) Що означає автоматичний виклик спеціальних методів?
13) Як можна визначити аргументи конструктора?
14) У чому полягає схожість між класом та структурою?
15) Як виконується визначення методу класу за його межами?
16) Що є специфікатором доступу до членів класу?
17) Для чого використовується специфікатор закритого доступу?
18) Який специфікатор доступу приймається за умовчуванням при оголошенні членів класу?
19) Якщо члени класу можна використовувати через об'єкт з основної програми, то який специфікатор доступу було задіяно при їх оголошенні?
20) Яке призначення мають класи?

13. СПАДКУВАННЯ

Навчальною метою розділу є ознайомлення читача зі спадкуванням властивостей одного класу іншим.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– створювати зв'язок спадкування між класами;

– використовувати специфікатори доступу при підключенні базового класу до похідного;

– розробляти ієрархічну структуру класів;

– передавати значення до аргументів базового класу через об'єкт похідного;

– перевантажувати методи базового класу у похідному;

– виконувати множинне спадкування;

– запобігати невизначеності при спадкуванні.

13.1. Спадкування при об'єктно-орієнтованому програмуванні

Спадкування є другим з трьох базових понять об'єктно-орієнтованого програмування. Так прийнято називати процес, при якому частина властивостей одного класу переходить до іншого. Перший клас іменують базовим, а другий – похідним. Сутність процесу полягає у тому, що при створенні об'єкта класу спадкоємця в ньому, окрім своїх членів, також створюється копія відкритих та захищених членів базового класу. Таким чином, похідний клас переймає від базового частину членів, а також має свої унікальні дані та методи. Цей підхід дозволяє поширювати можливості класів шляхом побудови ієрархічної структури, у якій кожен спадкоємець включає лише ті здібності, що необхідні для доповнення простого базового об'єкта до більш складного.

13.1.1. Створення похідних класів

Похідним називають клас, що успадковує від іншого відкриті та захищені члени. Формат оголошення похідного класу наведено нижче:

```
class <ім'я класу> : <специфікатор доступу> <ім'я базового класу>
{
        < оголошення функцій та змінних >
    <специфікатор доступу 1>:
        < оголошення функцій та змінних >
    <специфікатор доступу 2>
        < оголошення функцій та змінних >
   .
   .
   .
    <специфікатор доступу N>:
        < оголошення функцій та змінних >
}<список об'єктів>;
```

Різниця оголошень звичайного та похідного класів полягає лише у списку спадкування, що зазначається після імені через оператор ":". При підключенні базових класів використовуються вже відомі специфікатори доступу **private**, **public**, **protected**, що визначають вид доступу до копії членів.

Якщо базовий клас підключено через специфікатор **public**, то для методів похідного класу відкрито всі поля та методи базового класу, що мають відкритий та захищений види доступу. Через об'єкт класу можна застосовувати всі відкриті члени як похідного, так і базових класів. Робота з таким класом відповідає схемі, наведеній на рис. 13.1.

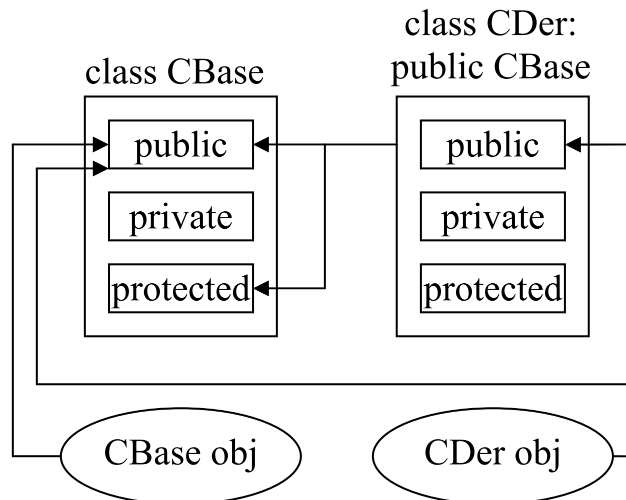

Рис. 13.1. Схематичне відображення доступу до членів базового та похідного класів при відкритому спадкуванні

Якщо ж базовий клас було підключено через специфікатор **private**, то методам похідного класу, як і при **public,** доступні всі відкриті та захищені члени, а для об'єкта лише відкриті поля та методи похідного класу. Робота з таким класом відповідає схемі, наведеній на рис. 13.2.

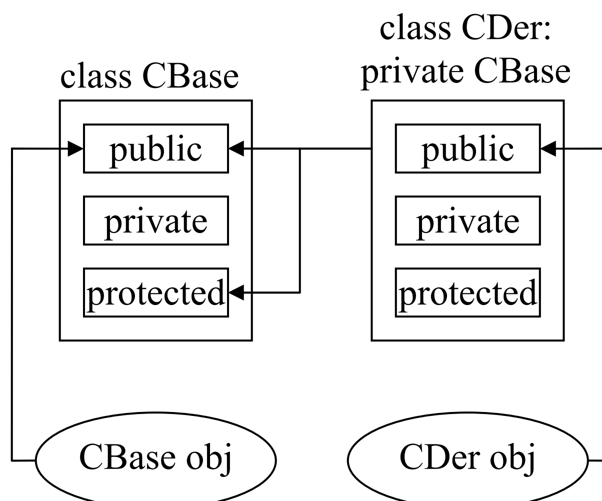

Рис. 13.2. Схематичне відображення доступу до членів базового та похідного класів при закритому спадкуванні

Підключення базового класу через специфікатор доступу **protected** відповідає схемі з закритим підключенням. Різниця з'являється лише при черговому спадкуванні.

Таким чином, специфікатори доступу при підключенні базового класу до похідного слугують для розширення інкапсуляції членів.

Розглянемо клас, що є похідним від розробленого раніше CSegment, і використовується для роботи з чотирикутником на площі:

```
//оголошення похідного класу та підключення до нього базового через private
class CQuaderangle : private CSegment
{
	public:				//члени з відкритим доступом
		int Xm[4];	//оголошення масиву для збереження абсцис точок
		int Ym[4];	//оголошення масиву для збереження ординат точок
//оголошення конструктора з аргументами
		CQuaderangle (int Ax, int Ay, int Bx, int By,
						int Cx, int Cy,int Dx, int Dy)
		{
//виклик методу для перевизначення координат вершин чотирикутника
			ReInitPoints(Ax,Ay,Bx,By,Cx,Cy,Dx,Dy);
		}
//оголошення конструктора без аргументів
		CQuaderangle ()
		{
//виклик методу для перевизначення координат вершин чотирикутника
			ReInitPoints(-10,20,10,20,100,-20,-100,-20);
		}
//оголошення методу для перевизначення координат вершин чотирикутника
		void ReInitPoints(int, int, int, int, int, int, int, int);
//оголошення методу, що перевіряє чи є фігура правильною
		bool IsEquilateral();
//оголошення методу, що повертає координату точки центра фігури
		void GetFigureCenter(int& x, int& y);
};

//визначення методу для перевизначення координат вершин чотирикутника
void CQuaderangle::ReInitPoints(int Ax, int Ay, int Bx, int By,
						int Cx, int Cy, int Dx, int Dy)
{
//визначення координати першої вершини чотирикутника
			Xm[0]=Ax;
			Ym[0]=Ay;
//визначення координати другої вершини чотирикутника
			Xm[1]=Bx;
			Ym[1]=By;
```

```
//визначення координати третьої вершини чотирикутника
                Xm[2]=Cx;
                Ym[2]=Cy;
//визначення координати четвертої вершини чотирикутника
                Xm[3]=Dx;
                Ym[3]=Dy;
}
//визначення методу, що первірє чи є фігура правильною
bool CQuaderangle::IsEquilateral()
{
//перевірка на те, чи має фігура чотири кути
        if((Xm[0]!=Xm[1] || Ym[0]!=Ym[1]) &&
            (Xm[0]!=Xm[2] || Ym[0]!=Ym[2]) &&
            (Xm[0]!=Xm[3] || Ym[0]!=Ym[3]) &&
            (Xm[1]!=Xm[2] || Ym[1]!=Ym[2]) &&
            (Xm[1]!=Xm[3] || Ym[1]!=Ym[3]) &&
            (Xm[2]!=Xm[3] || Ym[2]!=Ym[3]))
        {
//оголошення змінних для отримання координат
            int fv0,fv1,sv0,sv1;
//визначення координат точок протилежних вершин
            x0=Xm[0];
            y0=Ym[0];
            x1=Xm[2];
            y1=Ym[2];
//визначення координати центра першої діагоналі
            GetCenterPoint(fv0, fv1);
//визначення координат точок протилежних вершин
            x0=Xm[0];
            y0=Ym[0];
            x1=Xm[2];
            y1=Ym[2];
//визначення координати центра другої діагоналі
            GetCenterPoint(sv0, sv1);
//якщо точки центрів діагоналей збігаються
            if(fv0==sv0 && fv1==sv1)
            {
//визначення координати точки перетину діагоналей
                x0=fv0;
                y0=fv1;
//визначення координати точки першої вершини
                x1=Xm[0];
                y1=Ym[0];
//порахування суми довжин відрізка вздовж Ох та Оу
                sv0=GetXLength()+GetYLength();
```

```
//визначення координати точки другої вершини
                        x1=Xm[1];
                        y1=Ym[1];
//порахування суми довжин відрізка вздовж Ox та Oy
                        sv1=GetXLength()+GetYLength();
                        if(sv0==sv1)        //якщо отримані значення збігаються
//повернути підтвердження правильності фігури
                                return 1;
                        return 0;    //повернути спростування правильності фігури
                }
                return 0;            //повернути спростування правильності фігури
        }
        return 0;                    //повернути спростування правильності фігури
}

//визначення методу, що повертає координату точки центра фігури
void CQuaderangle::GetFigureCenter(int& x, int& y)
{
        x=y=0;          //визначення початкових значень для змінних
//цикл перебору всіх координат
        for(int i=0; i<4; i++)
        {
                x+=Xm[i];  //додати до змінної значення абсциси точки
                y+=Ym[i];  //додати до змінної значення ординати точки
        }
        x/=4; //порахувати абсцису точки центра
        y/=4; //порахувати ординату точки центра
}
```

Клас включає наступні члени: два поля масивів по чотири елементи, для збереження точок вершин чотирикутника, та методи CQuaderangle() *– конструктори класу для визначення початкових значень полів,* ReInitPoints() *– для перевизначення значень полів,* IsEquilateral() *– для первірки на те, що чотирикутник є правильним,* GetFigureCenter() *– для отримання координат точки центра фігури. Сутність успадкування для наведеного класу полягає в тому, що у тілі функції* IsEquilateral() *застосовані методи базового класу. Підключення* CSegment *до* CQuaderangle *відбувається за допомогою специфікатора доступу* **private**, *що робить члени базового класу закритими для застосування через об'єкти похідного класу.*

Важливо розуміти, що на основі процесу спадкування встановлюється лише однобічний зв'язок. Базовий клас не має жодного доступу до похідного.

Робота з похідними класами за допомогою змінних-покажчиків нічим не відрізнється від звичайних класів.

13.1.2. Виклик конструктора та деструктора базового класу

Спеціальні методи у похідних класах грають ту саму роль, що і у звичайних. Особливість стосується лише копії базового класу. Конструктор та деструктор завжди мають відкритий доступ, що приводить до їх обов'язкового копіювання у похідний клас. Виклик їх залишається автоматичним, але стає пов'язаним зі створенням об'єкта класу спадкоємця. Таким чином, для похідного класу існують два конструктори та два деструктори. Конструктор копії викликається перед конструктором самого класу, а деструктор копії після деструктора класу.

Щоб зрозуміти черговість автоматичних викликів спеціальних методів при спадкуванні, розглянемо приклад програми з використанням базового та похідного класів, код конструкторів та деструкторів котрих виводить на консоль відповідний до події статичний рядок:

```
#include <iostream>      //підключення бібліотеки потокового вводу/виводу
#include <iomanip>       //підключення бібліотеки застосування маніпуляторів

using namespace std;     //застосування простору імен std

class CBase              //оголошення базового класу
{
    public:              //відкриті члени класу
        CBase()          //оголошення та визначення конструктора
        {
//вивід статичного рядка на консоль
            cout<<"Call of CBase Constructor"<<endl;
        }
        ~CBase()         //оголошення та визначення деструктора
        {
//вивід статичного рядка на консоль
            cout<<"Call of CBase Destructor"<<endl;
        }
};
//оголошення похідного класу з відкритим підключенням базового
class CDer : public CBase
{
    public:              //відкриті члени класу
        CDer()           //оголошення та визначення конструктора
        {
//вивід статичного рядка на консоль
            cout<<"Call of CDer Constructor"<<endl;
        }
        ~CDer()          //оголошення та визначення деструктора
        {
//вивід статичного рядка на консоль
```

```
            cout<<"Call of CDer Destructor"<<endl;
        }
};

void main()         //оголошення та визначення головної функції
{
    CDer d;         //оголошення об'єкту похідного класу
}
```

Для роботи з потоковим вводом/виводом підключені бібліотеки iostream та iomanip. Код головної функції включає лише оголошення об'єкта похідного класу, що приводить до автоматичних викликів конструкторів. При досягненні закриваючої фігурної дужки об'єкт знищується, що відповідно викликає деструктори. Результат програми наступний:

```
Call of CBase Constructor
Call of CDer Constructor
Call of CDer Destructor
Call of CBase Destructor
```

Особливість викликів пов'язано з тим, що при реалізації любого методу похідного класу копія базового повинна вже існувати.

13.1.3. Визначення аргументів конструктора базового класу

Як вже було зазначено, конструктори достатньо часто мають аргументи, які визначаються при оголошенні об'єктів. Але визначити аргументи конструктора базового класу через об'єкт похідного неможливо. Це пов'язано з тим, що його оголошення, як і для звичайних класів, може слугувати лише задля визначення аргументів свого власного конструктора.

Визначення параметрів конструктору базового класу може бути здійснено лише за одним шляхом – через список ініціалізації конструктора похідного класу. Це єдине місце у програмі, де можна виконати виклик автоматичного методу конструктора.

Розглянемо приклад програми, в якій виконується визначення аргументу конструктора базового класу, що в свою чергу ініціює закрите поле.

```
#include <iostream>     //підключення бібліотеки потокового вводу/виводу
#include <iomanip>      //підключення бібліотеки застосування маніпуляторів

using namespace std;    //застосування простору імен std

class CBase             //оголошення базового класу
{
        int val;        //оголошення закритого поля
    public:             //відкриті члени класу
```

```
//оголошення та визначення конструктора, визначення поля
        CBase(int a) : val(a) //визначення поля через список ініціалізації
        {
        }
//оголошення та визначення методу для виводу значення поля на консоль
        void ShowValue()
        {
//вивід на консоль значення поля
            cout<<"val="<<val<<endl;
        }
};
//оголошення похідного класу з відкритим підключенням базового
class CDer : public CBase
{
    public:            //відкриті члени класу
//оголошення та визначення конструктора
        CDer(int b) : CBase(b) //виклик конструктора CBase
        {
        }
};

void main()          //оголошення та визначення головної функції
{
    int a;           //оголошення змінної для отримання значення від користувача
//вивід на консоль запиту до користувача
    cout<<"Enter value for private val:"<<flush;
    cin>>a;          //отримання введеного значення
    CDer d(a);   //оголошення об'єкта та визначення аргументу
//виклик копії методу базового класу виводу значення поля
    d.ShowValue();
}
```

Для роботи з потоковим вводом/виводом підключені бібліотеки iostream та iomanip. Базовий клас включає закрите поле для зберігання значення цілої змінної та два методи – конструктор і функцію виводу значення поля на консоль. Визначення поля виконується через список ініціалізації конструктора за допомогою значення, отриманого через аргумент. Похідний клас включає лише конструктор з аргументом, що використовується для передачі значення до конструктора базового класу, що викликається у списку ініціалізації. У головній функції виконується запит до користувача на ввід значення. Після вводу воно використовується для визначення аргументу об'єкта похідного класу. Значення закритого поля val виводиться на консоль за допомогою виклику копії методу базового класу.

13.2. Перевантаження методів базового класу у похідному

Перевантаження методів при спадкуванні дещо відрізняється від звичайного перевантаження. При спадкуванні є допустимим створити метод, який не тільки має однакове зі скопійованим ім'я, але й таку саму кількість та типи аргументів.

Розглянемо приклад програми, яка працює з похідним класом, що має перевантажений метод для розрахунку суми чисел від 1 до N з кроком в 1.

```
#include <iostream>       //підключення бібліотеки потокового вводу/виводу
#include <iomanip>        //підключення бібліотеки застосування маніпуляторів

using namespace std;      //застосування простору імен std

class CBase               //оголошення базового класу
{
        public:           //відкриті члени класу
//оголошення та визначення методу для розрахунку суми значень від 1 до val
                int GetSumFromTo(int val)
                {
//вивід на консоль статичного рядка, що визначає належність методу
                        cout<<"CBase function – "<<flush;
                        int sum=0;  //оголошення змінної та її початкове визначення
                        do          //цикл розрахунку суми
                        {
                                sum+=val--;        //додавання значення до суми
                        }while(val>0);      //умова виконання циклу
                        return sum;         //повернення отриманого значення
                }
};

class CDer : public CBase              //оголошення похідного класу
{
        public:                        //відкриті члени класу
//оголошення та визначення методу для розрахунку суми значень від 1 до val
                int GetSumFromTo(int val)
                {
                        if(val>1)     //якщо значення більше за 1
//повернення значення рекурсивного виклику методу похідного класу
                                return val+GetSumFromTo(val-1);
//вивід на консоль статичного рядка, що визначає належність методу
                        cout<<"CDer function – "<<flush;
                        return 1;     //повернення значення 1
                }
};
```

```
void main()          //оголошення та визначення головної функції
{
	int v;           //оголошення змінної для збереження значень користувача
	CDer d;          //оголошення об'єкта класу
//вивід на консоль запиту до користувача
	cout<<"Enter first value for calculating sum: "<<flush;
//отримання введеного значення
	cin>>v;
//вивід на консоль результату виклику методу похідного класу
	cout<<"Rezult is: "<<d.GetSumFromTo(v)<<endl;
//вивід на консоль запиту до користувача
	cout<<"Enter second value for calculating sum: "<<flush;
//отримання введеного значення
	cin>>v;
//вивід на консоль результату виклику методу базового класу
	cout<<"Rezult is: "<<d.CBase::GetSumFromTo(v)<<endl;
}
```

Код програми починається з підключення бібліотек для роботи з потоковим вводом/виводом. Потім виконується оголошення базового та похідного класів, кожен з яких має метод GetSumFromTo() *з одним цілим аргументом. У базовому класі сума розраховується за допомогою циклу з постумовою, а у похідному через рекурсивний виклик. У головній функції користувачеві виводяться два запроси на ввід значень, для кожного з яких сума розраховується через різні методи. При вводі значень* 5 *та* 7 *результат виконання програми буде наступним:*

```
Enter first value for calculating sum: 5
CDer function – Rezult is: 15
Enter first value for calculating sum: 7
CBase function – Rezult is: 28
```

Якщо через об'єкт похідного класу викликається перевантажений метод, що має повністю однакові оголошення, то компілятором автоматично обирається функція, яка не є скопійованою. Виклик іншої версії функції може бути виконано лише через операцію глобального дозволу "::", що зазначить клас, метод якого повинно бути викликано.

Перевантаження при спадкуванні використовується не лише для застосування різних підпрограм, але й для розширення можливостей методів, що вже існують. У наведеному вище прикладі для методу похідного класу застосовано властивість рекурсивного виклику. Цей код свідчить, що навіть з функцій у похідному класі перевантажений метод базового класу звичайним чином викликати неможливо, лише через операцію глобального дозволу.

Розглянемо, програму, що відповідає дещо зміненому попередньому прикладу. Вона не має двох реалізацій розрахунку суми. Перевантаження

методу у похідному класі виконується задля запобігання можливим помилкам користувача, пов'язаним з вводом від'ємних значень чи нуля.

```
#include <iostream>      //підключення бібліотеки потокового вводу/виводу
#include <iomanip>       //підключення бібліотеки застосування маніпуляторів

using namespace std;     //застосування простору імен std

class CBase              //оголошення базового класу
{
	public:              //відкриті члени класу
//оголошення та визначення методу для розрахунку суми значень від 1 до val
		int GetSumFromTo(int val)
		{
			int sum=0;  //оголошення змінної та її початкове визначення
			do          //цикл розрахунку суми
			{
				sum+=val--;        //додавання значення до суми
			}while(val>0);     //умова виконання циклу
			return sum;        //повернення отриманого значення
		}
};

class CDer : public CBase          //оголошення похідного класу
{
	public:                        //відкриті члени класу
//оголошення та визначення методу для розрахунку суми значень від 1 до val
		int GetSumFromTo(int val)
		{
			if(val>=1)   //якщо значення більше за одиницю
//повернути результат виконання методу базового класу
				return CBase::GetSumFromTo(val);
//вивід на консоль статичного рядка
			cout<<"Wrong value! – "<<flush;
			return 0;    //повернення значення 0
		}
};

void main()         //оголошення та визначення головної функції
{
	int v;          //оголошення змінної для збереження значень користувача
	CDer d;         //оголошення об'єкта класу
//вивід на консоль запиту до користувача
	cout<<"Enter first value for calculating sum: "<<flush;
	cin>>v;         //отримання введеного значення
```

```
//вивід на консоль результату виклику методу похідного класу
	cout<<"Rezult is: "<<d.CBase::GetSumFromTo(v)<<endl;
//вивід на консоль запиту до користувача
	cout<<"Enter second value for calculating sum: "<<flush;
//отримання введеного значення
	cin>>v;
//вивід на консоль результату виклику методу базового класу
	cout<<"Rezult is: "<<d.GetSumFromTo(v)<<endl;
}
```

Різниця у наведених кодах полягає у тому, що перевантажений метод не виконує окрему підпрограму, в його тілі перевіряється коректність отриманого значення і при відповідності викликається метод базового класу.

Перевантаження при спадкуванні використовується достатньо часто. Цей механізм дозволяє виконувати як дороботку вже існуючих методов, так і підвищує гнучкість ієрархічної структури класів.

13.3. Множинне спадкування і сутність невизначеності при ньому

Згідно з правилами спадкування, похідний клас може включати копії заразом декількох базових. Такий підхід називають множинним спадкуванням. Це значно поширює можливості ієрархічного спадкування, бо один складний об'єкт може складатися з необхідної кількості простих. Крім того, прості класи є легкими в реалізації, тому для програміста при розробці складного проекту достатнім є лише виконати у правильній послідовності зв'язки між членами об'єктів.

Наприклад, за множинним наслідуванням може бути створено ієрархію класів для роботи з геометричними фігурами у просторі за схемою на рис. 13.3.

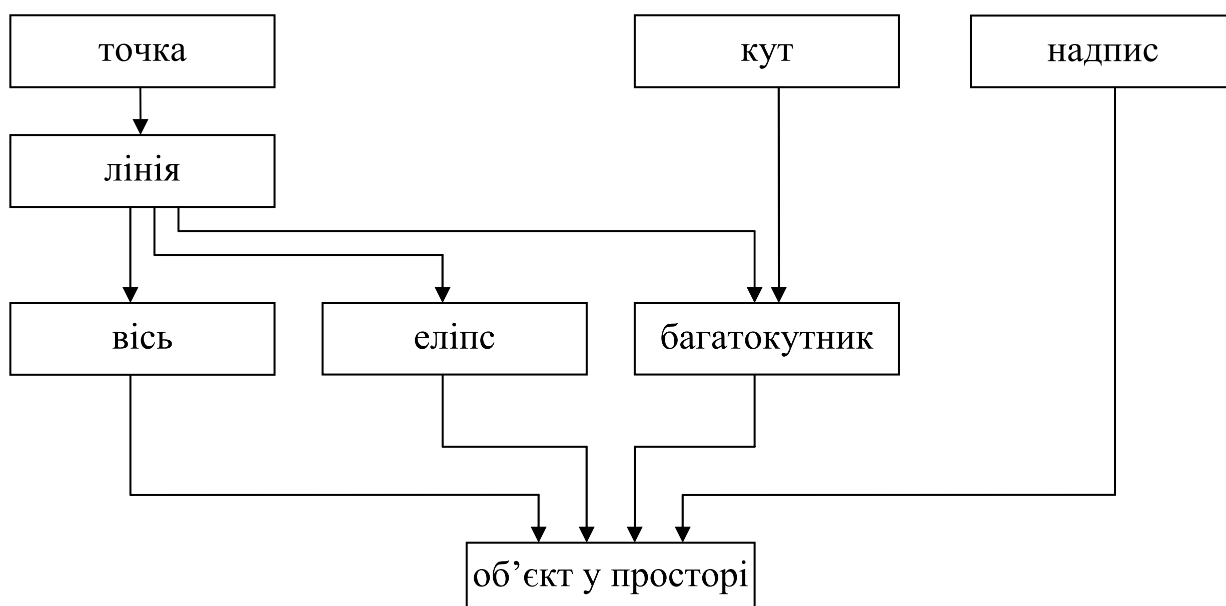

Рис. 13.3. Ієрархія класів роботи з графічними об'єктами на площині

Множинне спадкування виконується за форматом оголошення похідного класу. Базові класи підключають за допомогою застосування специфікаторів доступу через оператор "кома":

class *<ім'я класу>* : *<специфікатор доступу 1> <ім'я 1-го базового класу>*,
<специфікатор доступу 2> <ім'я 2-го базового класу>. . .

Так, клас CDer, що є похідним від CA, CB та CC може бути оголошений за наступним прикладом:

class CDer : **public** CA, **private** CB, **public** CC
{};

Перевагою множинного спадкування є те, що воно значно розширює об'єктно-орієнтований підхід при написанні програмного забезпечення. Недоліком можна вважати деякі додаткові витрати на планування, щоб запобігти невизначеності.

Невизначеністю при множинному спадкуванні називають ситуацію, коли компілятор не може без додаткового визначення зрозуміти, яку з копій перевантажених методів необхідно підставити у місце виклику.

Частіше за все невизначеність виникає у випадку, коли похідний клас спадкує від двох базових методи з однаковими іменами. Таким чином, у ньому існують заразом дві копії відповідної функції і, при її виклику, компілятору є незрозумілим, яку з підпрограм необхідно використати (рис. 13.4).

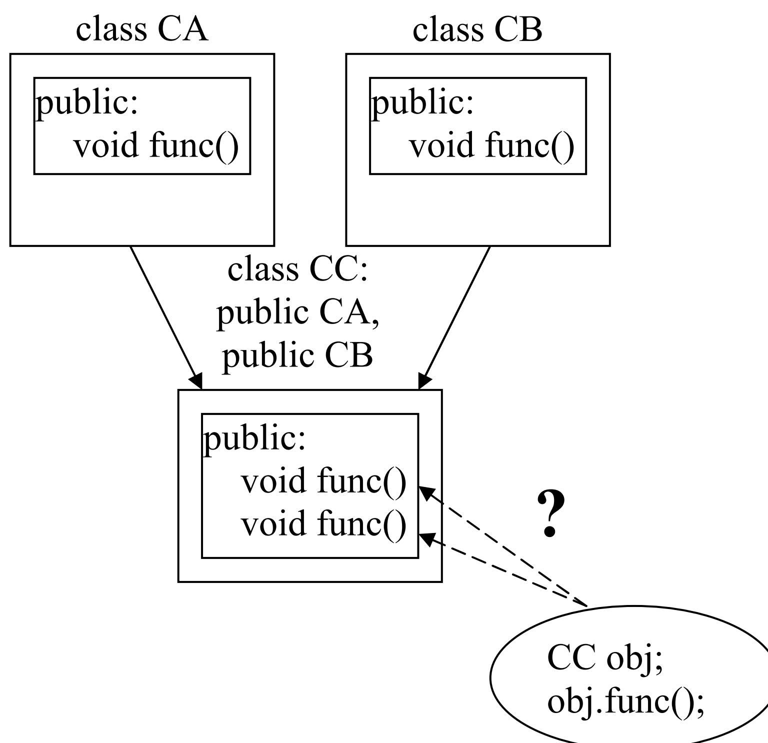

Рис. 13.4. Схематичне відображення невизначеності при множинному спадкуванні

Одним з шляхів вирішення невизначеності є вже розглянута операція глобального визначення – "::". Якщо при всіх викликах перевантажених методів буде використано ім'я відповідного класу, то перед компілятором не буде стояти необхідність вибору. Наприклад, для наведеної вище схеми:

```
CC obj;           //оголошення об'єкта похідного класу
obj.CA::func();   //виклик методу, що є копією з  CA
obj.CB::func();   //виклик методу, що є копією з  CB
```

Ще один шлях вирішення невизначеностей полягає у застосуванні віртуалізації. Його буде розглянуто у розділі "Поліморфізм".

Висновки

У даному розділи було розглянуто наступні основні питання:

– спадкування класів;

– обмеження доступу при спадкуванні;

– переваження методів при спадкуванні;

– множинне спадкування;

– запобігання невизначеності при множинному спадкуванні.

Контрольні питання

1) Дайте визначення поняттю спадкування.

2) Назвіть та охарактеризуйте класи, що пов'язані через спадкування?

3) Як виконується підключення базового класу?

4) У чому різниця між спадкуванням через специфікатори відкритого та закритого доступу?

5) Як виконується виклик методів похідного класу через змінну-покажчик?

6) В чому полягає особливість автоматичного виклику конструктора та деструктора базового класу при оголошенні об'єкта похідного класу?

7) Які члени базового класу і при якій умові доступні через об'єкт похідного класу?

8) Як виконується визначення аргументів конструктора базового класу через об'єкт похідного класу?

9) В чому полягає особливість списків ініціалізації стосовно до конструктора базового класу?

10) Дайте визначення поняттю перевантаження методів при спадкуванні.

11) Поясніть, чому при виклиці методу, що було визначено у базовому класі та перевантажено при спадкуванні, через об'єкт похідного класу реалізується лише функція похідного класу?

12) Як через об'єкт похідного класу викликати перевантажений метод базового класу?

13) У чому полягає призначення перевантаження методів при спадкуванні?

14) Дайте визначення множинному спадкуванню.

15) Що надає програмісту використання множинного спадкування?

16) Яку ситуацію називають невизначеністю при множинному спадкуванні?

17) Яка ситуація, що призводить до невизначеності при множинному спадкуванні, є найбільш поширеною?

18) Як вирішується невизначеність при множинному спадкуванні?

19) Які переваги програмісту дає використання спадкування при розробці складних об'єктно-орієнтованих проєктів?

20) Доступ до яких членів похідного класу можна отримати через об'єкт базового класу при його підключенні через специфікатор доступу **protected**?

14. ВІРТУАЛЬНІ, ДРУЖНІ ТА СТАТИЧНІ ФУНКЦІЇ

Навчальною метою розділу є ознайомлення читача з поліморфізмом, дружніми функціями і класами, та статичними членами.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– реалізовувати поліморфізм на базі віртуальних функцій;

– застосовувати чисті віртуальні функції для створення ієрархічної структури класів;

– виконувати виклик деструктора базового при роботі з динамічним об'єктом похідного класу;

– запобігати невизначеності при спадкуванні за допомогою віртуальних функцій та класів;

– застосовувати дружні функції та класи для роботи з закритими та захищеними членами класів;

– використовувати статичні члени класів для взаємодії між об'єктами.

14.1. Поліморфізм

Поліморфізм означає мінливість чи здібність до перетворювань. Для об'єктно-орієнтованого програмування поліморфізм – це властивість виклику однойменних методів похідних від одного базового, класів через змінну-покажчик на клас, що спадкується. Наприклад, припустимо при наявності ієрархічної побудови класів для відображення графічних фігур класи CTriangle, CSquare та CCircle є похідними від базового класу CFigure. В кожному з них визначено метод Draw(), що дозволяє зображувати фігуру на формі. Зручність застосування поліморфізму полягає у тому, що, створивши масив fig[3] змінних-покажчиків на клас CFigure і доповнивши його значеннями адрес об'єктів похідних класів, можна виконати відображення всіх трьох фігур за допомогою нижче наведеного лістинга:

```
for(int i=0; i<3; i++)          //цикл перебору індексів масиву
    f[i]->Draw();               //виклик методу об'єкта
```

Таким чином, опрацювання однойменних методів будь-якої кількості похідних від одного базового класів можна реалізувати, застосувавши лише один цикл, в якому будуть перебиратися адреси існуючих об'єктів.

Але використання властивості поліморфізму в об'єктно-орієнтованому програмуванні потребує, щоб базовий клас теж мав однойменний метод з похідними. Окрім того, метод базового класу повинен бути оголошений віртуальним. Лише в цьому випадку компілятор мови C++ зрозуміє, що є потреба перейти від раннього до пізнього зв'язування, яке і відкриє можливість застосування властивості поліморфізму.

14.1.1. Віртуальні функції

Слово віртуальний (virtual – англ.) означає: той, що є видимим, але не існуючим насправді. Віртуальною функція стає у тому випадку, коли при її

оголошенні та визначенні було використано службове слово **virtual**. Формат оголошення віртуальної функції зазначено нижче:

virtual <*тип даних* > <*ім'я функції* > (<*список параметрів*>);

Розглянемо сутність використання віртуальних функцій на наступному прикладі: створимо базовий клас та два похідних від нього; визначимо в цих класах однойменні відкриті методи, що виконують різні дії; у функції main() реалізуємо виклик всіх трьох методів через єдину змінну-покажчик.

```
#include <iostream>              //підключення бібліотеки вводу/виводу
using namespace std;             //застосування простору імен
class CBase                      //оголошення базового класу
{
        public:                  //відкриті члени класу
        //оголошення та визначення методу виводу значення
                void show_num()
                {
                        cout<<"0"<<endl;        //вивід константного рядка
                }
};

class CDer1 : public CBase       //оголошення класу спадкоємця
{
        public:                  //відкриті члени класу
//оголошення та визначення методу виводу значення
                void show_num()
                {
                        cout<<"1"<<endl;        //вивід константного рядка
                }
};

class CDer2 : public CBase       //оголошення класу спадкоємця
{
        public:                  //відкриті члени классу
//оголошення та визначення методу виводу значення
                void show_num()
                {
                        cout<<"2"<<endl;        //вивід константного рядка
                }
};

void main()                      //оголошення та визначення головної функції
{
        CBase *pB,base;          //оголошення змінної-покажчика та об'єкта
```

```
    CDer1 der1;             //оголошення об'єкта
    CDer2 der2;             //оголошення об'єкта
    pB=&base;         //визначення змінної-покажчика адресою об'єкта
    pB->show_num();         //виклик методу
    pB=&der1;         //визначення змінної-покажчика адресою об'єкта
    pB->show_num();         //виклик методу
    pB=&der2;         //визначення змінної-покажчика адресою об'єкта
    pB->show_num();         //виклик методу
}
```

Класи CDer1 *та* CDer2 *є похідними від* CBase*. В кожному з трьох класів створено метод* show_num()*, що є перезавантаженим. Завдячуючи тому, що методи виводять на консоль унікальні числа, можна відстежити який з них було реалізовано. У головній функції послідовно до змінної-покажчика на* CBase *записуються адреси об'єктів всіх трьох класів та виконується виклик методу* show_num().

Треба зазначити, що таке написання програмного коду не є помилкою. Раніше було сказано, що неприпустимо визначати змінну-покажчик на один тип даних адресою змінної іншого типу даних, але об'єкти, що відповідають базовому та похідним від нього класам, компілятором вважаються змінними одного типу даних. Ця особливість об'єктно-орієнтованого програмування і дає можливість програмістам застосовувати властивість поліморфізму.

При виконанні програмного коду прикладу на консоль буде виведено наступний стовпець з цифр:

```
0
0
0
```

Це зовсім не відповідає очікуваному результатові. Компілятор не зміг визначити різниці між адресами об'єктів і кожного разу було виконано метод базового класу.

Внесемо до нашої програми невеличку зміну – нехай метод show_num() базового класу стане віртуальним. Його визначення буде мати наступний вигляд:

```
//оголошення та визначення віртуального методу виводу значення
virtual void show_num()
{
    cout<<"0"<<endl;        //вивід константного рядка
}
```

Нове виконання програмного коду виведе на консоль наступний стовпець:

0
1
2

14.1.2. Раннє та пізнє зв'язування

Таким чином, якщо метод базового класу, що перезавантажено у похідних класах, буде оголошено як віртуальний, то компілятор починає розуміти різницю між адресами об'єктів. Хоча насправді це лише видимість. Через змінну-покажчик, яку визначено адресою, компілятор не може дізнатися про тип об'єкта. Підхід, що застосовується у цьому випадку, називають пізнім зв'язуванням. Протилежне до нього раннє зв'язування є класичним при компіляції програм, в яких відсутні віртуальні методи чи класи. При ньому компілятор виконує зв'язування змінної-покажчика з методами та полями того класу, що зазначено при його оголошенні. Цей підхід було обрано при першому виконанні наведеного прикладу. На відміну від раннього, при пізньому зв'язуванні компілятор відкладає встановлення зв'язків до запуску програми, під час якого через існуючу явним чином адресу об'єкта застосовується відповідна функція. Це сталося під час другого виконання прикладу. Раннє та пізнє зв'язування також часто називають статичним та динамічним.

14.1.3. Чисті віртуальні функції та абстрактні класи

З наведеного вище прикладу можна зробити висновок, що при відтворенні поліморфізму у програмах базовий клас необов'язково повинен мати реалізацію для віртуальних методів. Зручно створити базовий клас з безліччю порожніх віртуальних методів для пізнього зв'язування, а всі особливості їх реалізації, пов'язані з призначенням, перекласти на перезавантажені у похідних класах однойменні методи. Таким чином, базовий клас з окремого елемента формування при застосуванні властивості поліморфізму перетворюється на шаблонну основу для організації ієрархічної структури похідних класів.

Щоб перейти до зазначеного застосування базового классу, використовуються чисті віртуальні функції. Віртуальна функція стає чистою віртуальною в тому випадку, коли при її оголошенні було застосовано наступний формат:

virtual <*тип даних* > <*ім'я функції*>(<*список параметрів*>)=0;

"=0" не пов'язано з присвоюванням значення нуля, це лише форма запису, що визначає віртуальну функцію як чисту віртуальну.

Визначати чисті віртуальні функції не треба, бо вони у програмі не можуть бути реалізовані. Спроба їх виклику призведе до помилки при компіляції. Окрім того, оголошення лише однієї чистої віртуальної функції має вплив на весь клас цілком. Всі методи такого класу автоматично стають чистими віртуальними, а він сам перетворюється на абстрактний. Для таких класів об'єктів існувати не може, лише змінні-покажчики. Основне призначення

абстрактних класів полягає у формуванні на їх основі ієрархічних структур похідних класів для реалізації властивості поліморфізму.

14.2. Віртуальний деструктор

Якщо при написанні програм програміст застосовує властивість поліморфізму, коли працює з динамічно створеними об'єктами похідних класів і хоче, щоб при вивільненні пам'яті було викликано деструктор похідного класу, то деструктор базового класу обов'язково повинен бути оголошений як віртуальний.

Розглянемо приклад:

```
#include <iostream>                    //підключення бібліотеки вводу/виводу

using namespace std;                   //застосування простору імен

class CBase                            //оголошення базового класу
{
    public:                            //відкриті члени класу
        ~CBase()                       //оголошення та визначення деструктора
        {
            cout<<"Base"<<endl;    //вивід константного рядка
        }
};

class CDer : public CBase
{
    public:                            //відкриті члени класу
        ~CDer()                        //оголошення та визначення деструктора
        {
            cout<<"Derived"<<endl;         //вивід константного рядка
        }
};

void main()                            //оголошення та визначення головної функції
{
    CBase *pB=new CDer;                //динамічне створення об'єкта
    delete pB;                         //вивільненні пам'яті
}
```

В програмі створено базовий клас CBase *та похідний від нього клас* CDer. *В кожному з класів визначено деструктор. При видаленні об'єкта відповідний деструктор виводить на консоль назву класу, до якого належить. У головній функції виконується динамічне створення об'єкта класу* CDer *та запис його адреси у змінну-покажчик на клас* CBase. *Наступним рядком об'єкт видаляється.*

Результатом виконання наведеної програми є:

```
Base
```

Це означає, що деструктор похідного класу було проігноровано. Такий підхід може призвести до значних помилок роботи програми.

Змінимо простий деструктор базового класу на віртуальний:

```
virtual ~CBase()            //оголошення та визначення віртуального  деструктора
{
        cout<<"Base"<<endl;            //вивід константного рядка
}
```

Запуск нової версії програми приведе до появи на консолі рядків:

```
Derived
Base
```

Цей результат повністю відповідає поставленій задачі.

Зазначимо, що необхідність у віртуалізації деструкторів зникає, коли об'єкти створено статичним шляхом.

14.3. Віртуальні базові класи

Ще одним призначенням віртуалізації, але вже не пов'язаним з поліморфізмом, є усунення невизначеності при множинному спадкуванні. Для цієї задачі використовуються віртуальні базові класи.

Таким чином, наведену на рис. 14.1 схему множинного спадкування, можливо реалізувати через віртуальні класи без невизначеності.

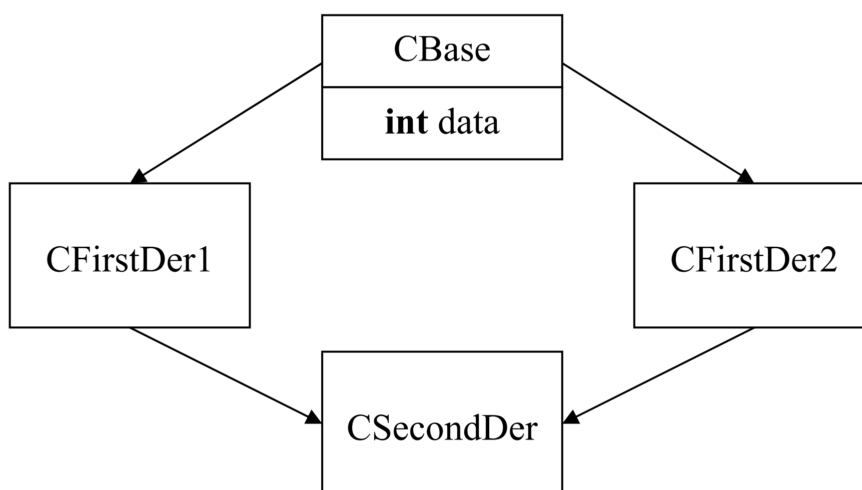

Рис. 14.1. Схема множинного спадкування

```
class CBase                                   //оголошення базового класу
{
```

```
	public:				//відкриті члени класу
		int data;			//оголошення цілої змінної
};

class CFirstDer1 : virtual public CBase	//оголошення класу спадкоємця
{
};

class CFirstDer2 : virtual public CBase	//оголошення класу спадкоємця
{
};

//оголошення класу спадкоємця
class CSecondDer : public CFirstDer1, public CFirstDer2
{
	public:				//відкриті члени класу
		void func(int i)		//оголошення та визначення методу
		{
			data=i;		//визначення поля класу
		}
};
```

В цьому фрагменті програмного коду невизначеність відсутня, бо при підключенні базового класу до обох спадкоємців було зазначено, що він є віртуальним. Такий підхід не впливає на базовий клас, а лише говорить компілятору, що для всіх похідних класів буде існувати лише єдина копія класу, що було успадковано.

14.4. Дружні функції та дружні класи

Застосування інкапсуляції в об'єктно-орієнтованому програмуванні призводить до того, що глобальні функції, що не є методами класів, не спроможні мати доступ до їх полів. Іноді таке нерушиме правило може призвести до деяких незручностей при програмуванні, бо виникає необхідність у застосуванні закритих чи захищених полів класу у глобальній функції. В цьому випадку на допомогу приходить властивість створення дружніх до вказаного класів і функцій. Такі класи і функції можуть виконувати операції з інкапсульованими даними та методами без явного формування залежності від вказаного класу.

Зазначення дружності виконується за допомогою службового слова **friend**.

Для створення вказівки компілятору про те, що функція чи клас для поточного класу буде виступати у якості дружньої, треба в його тілі скористатися одним з наступних форматів:

friend *<тип даних > <ім'я функції >*(*<список параметрів >*);

friend class <*ім'я класу* >;

Розглянемо программу, у якій застосовано дружність функції ffunc() та класу CFriend до класу CTest з закритим полем data та методом inc().

```
#include <iostream>              //підключення бібліотеки вводу/виводу
using namespace std;             //застосування простору імен

class CTest                      //оголошення класу
{
        private:                 //закриті члени класу
                int data;        //оголошення поля збереження даних
//оголошення та визначення методу інкрементування поля
                void inc()
                {
                        data++;  //інкрементування значення поля
                }
        public:                  //відкриті члени класу
//оголошення та визначення конструктора
                CTest() : data(5)   //визначення поля за умовчуванням
                {
                }
                friend void ffunc(CTest);      //оголошення дружньої функції
                friend class CFriend;          //оголошення дружнього класу
};
//визначення класу, що є дружнім
class CFriend
{
        public:                                //відкриті члени класу
                void fmetod(CTest t)           //оголошення та визначення методу
                {
//вивід значення поля на консоль
                        cout<<"data is:"<<t.data<<endl;
                        t.inc();               //інкрементування значення поля
//вивід значення поля на консоль
                        cout<<"data+=1 is: "<<t.data<<endl;
                }
};

void ffunc(CTest t)              //визначення функції, що є дружньою до класу
{
        cout<<"data is:"<<t.data<<endl;        //вивід значення поля на консоль
        t.inc();                               //інкрементування значення поля
        cout<<"data+=1 is: "<<t.data<<endl;//вивід значення поля на консоль
}
```

```
void main()                     //оголошення та визначення головної функції
{
    CTest a;                    //оголошення об'єкта класу
    ffunc(a);                   //виклик дружньої функції
    CFriend f;                  //оголошення об'єкта дружнього класу
    f.fmetod(a);                //виклик методу дружнього класу
}
```

Клас CTest, *окрім поля datata метода* inc(), *має конструктор, у якому виконується ініціалізація data значенням 5, та рядки, що повідомляють компілятор про необхідність відкрити функції* ffunc() *та класу* CFriend *повний доступ як дружнім. Ці рядки не служать у якості оголошення відповідної функції та класу. Оголошення та визначення виконуються нижче. Функція* ffunc() *та єдиний метод класу* CFriend *мають однакову реалізацію: вивід на консоль двох рядків, поточного значення змінної* data *та її значення після інкрементування. У головній функції створюється об'єкт* a *класу* CTest, *а потім викликаються* ffunc() *та* fmetod().

Запуск програми приводить до появи на консолі наступного результату роботи:

```
data is: 5
data+=1 is: 6
data is: 5
data+=1 is: 6
```

Таким чином, використання властивості дружності призводить до ігнорування інкапсуляції функціями та класами, що зазначені як дружні. Але цей підхід треба застосовувати обережно, бо зловживання дружністю може порушити стійкість структури об'єктно-орієнтованої програми.

14.5. Статичні поля та функції

При роботі з об'єктами класу іноді виникає необхідність в організації між ними зв'язків, що базуються на використанні спільних змінних. Цю функцію можуть виконувати глобальні змінні, але такий підхід не завжди є зручним. По-перше є недоліки застосування глобальних змінних, а по-друге порушується незалежність об'єктів класів від програми в цілому.

Правильне вирішення такої задачі полягає у використанні статичних членів класу. Їх застосування дозволяє організовувати зв'язок між окремими об'єктами та виконувати одночасну ініціалізацію, що є важливим при необхідності зміни параметрів налаштування об'єктів.

Особливістю статичних членів класу є те, що для всіх об'єктів існує лише єдина їх версія. Це наглядно демонструє схема на рис. 14.2.

CAny

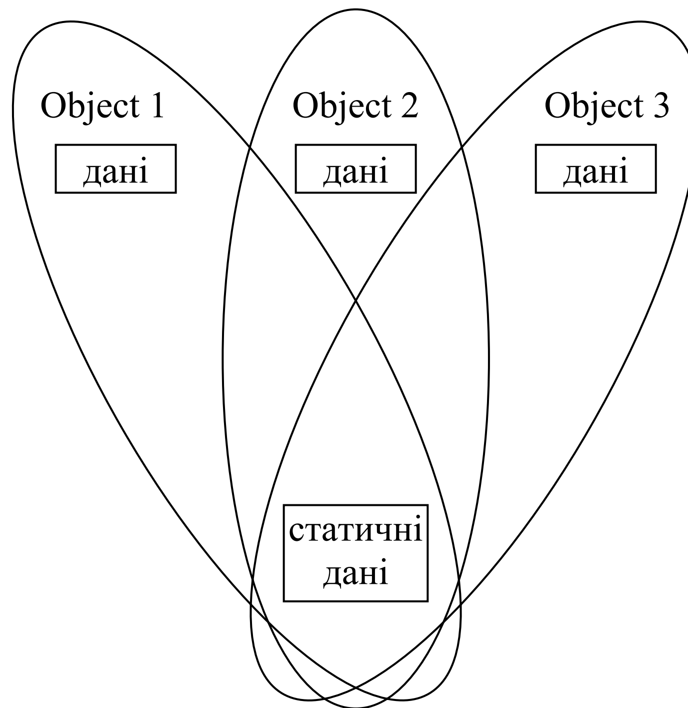

Рис. 14.2. Статичні дані відносно об'єктів класу

Незалежно від кількості об'єктів статичні члени класу завжди будуть подані в одному екземплярі. У програмі для оголошення статичних полів та методів використовуються нижче наведені формати:

static <*тип даних* ><*ім'я поля* >;
static <*тип даних* ><*ім'я методу* >(<*список параметрів* >);

Розглянемо приклад, у якому реалізовано лічильники кількості створених з початку роботи об'єктів для вказівки унікальних ідентифікаторів та поточної кількості об'єктів.

```
#include <iostream>                //підключення бібліотеки вводу/виводу

using namespace std;               //застосування простору імен

class CAny                         //оголошення класу
{
        private:                   //закриті члени класу
//оголошення статичного поля лічильника загальної кількості об'єктів
                static int tcounter;
//оголошення статичного поля лічильника поточної кількості об'єктів
                static int ccounter;
                int ID;            //оголошення поля ідентифікатора об'єкта
```

```
    public:
        CAny()                  //оголошення та визначення конструктора
        {
            tcounter++;         //інкрементування загального лічильника
            ccounter++;         //інкрементування поточного лічильника
            ID= tcounter;       //визначення ідентифікатора об'єкта
        }
        ~CAny()                 //оголошення та визначення деструктора
        {
            ccounter--;         //декрементування поточного лічильника
        }
//оголошення та визначення методу виводу ідентифікатора
        void showID()
        {
//вивід ідентифікатора об'єкта
            cout<<"Object ID is "<<ID<<endl;
        }
//оголошення та визначення статичного методу виводу поточного лічильника
        static void showCounterV()
        {
//вивід значення поточного лічильника
            cout<<"Objects: "<<ccounter<<endl;
        }
};

int CAny::ccounter=0;           //визначення статичного поля за умовчуванням
int CAny::tcounter=0;           //визначення статичного поля за умовчуванням
void main()
{
    CAny *pA1,*pA2;             //оголошення змінних-покажчиків на клас
    pA1=new CAny;               //динамічне створення об'єкта
    delete pA1;                 //вивільнення пам'яті
    pA2=new CAny[5];            //динамічне створення масиву об'єктів
    pA1=new CAny;               //динамічне створення об'єкта
    pA1->showID();              //виклик методу виводу ідентифікатора
    (pA2+3)->showID();          //виклик методу виводу ідентифікатора
//виклик статичного методу виводу поточного лічильника
    CAny::showCounterV();
    delete pA1;                 //вивільнення пам'яті
    delete [] pA2;              //вивільнення пам'яті
//виклик статичного методу виводу поточного лічильника
    CAny::showCounterV();
}
```

У програмі створено клас CAny*, що включає два статичні поля лічильника; поле унікального ідентифікатора об'єкта; конструктор, що інкрементує лічильники та ініціює змінну ідентифікатор; деструктор, який дектрементує лише лічильник поточної кількості; метод, що виводить на консоль ідентифікатор об'єкта та статичний метод, що відображає на консолі поточну кількість об'єктів. У головній функції оголошено дві змінні-покажчика на клас* CAny*. Динамічним шляхом об'єкти створюються та видаляються, на консоль виводяться зазначені ідентифікатори та поточна кількість об'єктів.*

Результат роботи наведеної програми буде полягати у виводі на консоль наступних чотирьох рядків:

Object ID is 7
Object ID is 5
Objects: 6
Objects: 0

Таким чином, завдячуючи загальному лічильнику, кожен об'єкт при створенні отримує унікальний ідентифікатор, відповідає черзі виникнення в процесі виконання програми. Поточний лічильник зберігає кількість об'єктів, що існує на вказаний час.

Треба зазначити, що для першої ініціалізації статичних полів необхідно вказувати глобальне їх визначення перед використанням згідно з наступним форматом:

<*тип даних* ><*ім'я класу* >::<*ім'я поля* >=<*значення*>;

Таке визначення через операцію глобального дозволу не є новим оголошенням поля. Воно вказує компілятору, що є пам'ять, яка закріплена за зазначеним полем, буде спільною для всіх об'єктів.

Основним призначенням для статичних методів класу є можливість їх виклику незалежно від створених об'єктів через операцію глобального дозволу. Окрім того, такий підхід не забороняє викликати їх класичним шляхом. Останній рядок наведеного прикладу реалізує виклик статичного методу showCounterV() при відсутності будь-яких об'єктів класу CAny.

Застосування статичних членів класів не залежить від обраного, динамічного чи статичного, шляху створення об'єктів.

Висновки

У даному розділи було розглянуто наступні основні питання:

– поліморфізм;
– віртуальні функції;
– абстрактні класи;
– дружні функції та класи;
– статичні члени класів.

Контрольні питання

1) Що таке поліморфізм?

2) Яку функцію називають віртуальною?

3) У чому полягає призначення віртуальних функцій?

4) Яка різниця між раннім та пізнім зв'язуванням?

5) Яку віртуальну функцію називають чистою?

6) Для чого потрібні чисті віртуальні функції?

7) Який клас називають абстрактним?

8) Чому при роботі зі спадкоємцем через змінну-покажчик деструктор базового класу обов'язково повинен бути віртуальним?

9) Як за допомогою віртуальних класів вирішується невизначеність при множинному спадкуванні?

10) Яких правил необхідно дотриматись, щоб реалізувати поліморфізм на основі віртуальних функцій?

11) Які функції називають дружніми?

12) Як вказати, що зазначена функція буде дружньою для класу?

13) Для чого застосовують дружні класи?

14) У чому полягає спостереження при застосування властивостей дружності?

15) Які поля класу називають статичними?

16) У чому полягає сутність застосування статичних полів класу?

17) Як визначити статичне поле за умовчуванням?

18) Які методи класу називають статичними?

19) Як виконується виклик статичних методів класу?

20) Яке основне призначення мають статичні методи класів?

15. ШАБЛОНИ ФУНКЦІЙ І КЛАСІВ

Навчальною метою розділу є ознайомлення читача з механізмом створення шаблонів функцій та класів.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– створювати та використовувати шаблони функцій;

– застосовувати шаблони класів для написання універсальних програмних кодів.

15.1. Механізм шаблонів

Під час розробки великих і складних проектів на мові С++, часто виникає необхідність використання функцій, які виконують одну задачу, але використовують дані різних типів.

В мові С для виконання схожої задачі створюється ціла група функцій, імена котрих вказують с даними якого типу працює саме цей екземпляр. Наприклад, для того, щоб отримати модуль числа, необхідно створити наступні функції:

для типу **char** – cabs()
для типу **int** – iabs ()
для типу **long int** – labs()
для типу **float** – fabs()
для типу **double** – dabs()

Тіла цих функцій були б повністю ідентичні для аргументу **X**, що передається:

```
{return (x<0) ? (–x) : x ;}                //повернути значення за модулем
```

В мові С++ вирішення цієї задачі має інший вигляд, тому що в цій мові виникає механізм перегруження функцій. Таким чином, була б розроблена така сама група функцій, які мають однакове ім'я abs(). Механізм перегруження спрощує виклик такої функції, але ніяк не розмір програмного коду, який необхідно написати для визначення перегруженних функцій. Наприклад, для знаходження суми двох чисел, необхідно створити функцію sum(), яка має наступні перегруженні визначення:

```
//оголошення та визначення функції розрахунку суми двох значень типу char
double Sum (char a, char b)
{return a + b ;}                           //повернути значення суми
//оголошення та визначення функції розрахунку суми двох значень типу int
double Sum (int a, int b)
{return a + b ;}                           //повернути значення суми
//оголошення та визначення функції розрахунку суми двох значень типу long int
double Sum (long int a, long int b)
```

```
{return a + b ;}                    //повернути значення суми
//оголошення та визначення функції розрахунку суми двох значень типу float
double Sum (float a, float b)
{return a + b ;}                    //повернути значення суми
//оголошення та визначення функції розрахунку суми двох значень типу double
double Sum (double a, double b)
{return a + b ;}                    //повернути значення суми
```

Як бачимо із наведеного вище прикладу, тіла функцій, які використовуються для виконання певної задачі з даними різних типів, ідентичні. Це можна сприймати і як перевагу, і як недолік. Загалом написання групи таких функцій зводиться до пропису рядків визначення і одного тіла. Далі для всіх визначень, що залишилися, тіло можна просто скопіювати. Однак, що буде якщо тіло функції не настільки просте, як в наведених прикладах, і займає десять або двадцять рядків? Ваша програма під час пропису кожної наступної функції групи буде збільшуватися в розмірах. А що буде, якщо в процесі написання програми ви знайдете помилку в тілі такої функції? Це призведе до того, що її прийдеться виправляти не один раз, а у всіх екземплярах. Таким чином, використання групи функцій для виконання окремої задачі з даними різних типів є не стільки складним, як клопітким процесом.

В мову С++ закладено засіб, який може полегшувати таку задачу. Це механізм застосування шаблонів. Його може бути використано як для функцій, так і для класів.

15.1.1. Шаблон функції

Оголошення шаблону виконується за наступним форматом:

template < **class** <*ім'я шаблонного типу даних* > >

Приклад оголошення шаблону функцій для обчислення різниці двох чисел наведено нижче:

```
template < class T >     //зазначення шаблону типу даних T
T Sub (T a, T b)         //оголошення та визначення функції розрахунку різниці
{
      return a – b;      //повернути результат розрахунку різниці
}
```

Слово **template** є службовим і використовується для оголошення шаблонного типу даних. Ключове слово **class**, взяте у кутові дужки, може бути замінене на **type**.

Коли компілятор зустрічає слово **template**, то практично нічого не виконує. Шаблон не викликає генерацію будь-якого коду, оскільки компілятор

ще не може знати, який об'єм пам'яті необхідно виділити для цієї функції. Шаблон запам'ятовується для наступного використання.

Генерація коду виконується в момент виклику. Виклик такої функції називається реалізація шаблону. При цьому кожен реалізований шаблон називають шаблонною функцією.

Приклад створення і використання шаблона функції для знаходження добутку двох чисел:

```
#include <iostream>              //підключення бібліотеки потокового вводу/виводу
using namespace std;             //застосування простору імен std

template <class T>               //зазначення шаблону типу даних T
//оголошення та визначення шаблонної функції розрахунку добутку
T Mult (T a, T b)
 {
       return  a*b ;             //повернення значення добутку
}
void main()                      //оголошення та визначення головної функції
{
       int  c1 = 3;              //оголошення та визначення цілої змінної
       int  c2 =  -7;            //оголошення та визначення цілої змінної
       float  f1 = 2.3; //оголошення та визначення змінної з плаваючою точкою
       float  f2 = 4.1; //оголошення та визначення змінної з плаваючою точкою
       unsigned int  i1 = 321;          //оголошення та визначення цілої змінної
       unsigned int  i2 = 2;            //оголошення та визначення цілої змінної
//вивід на консоль результату роботи функції для цілих змінних
       cout << c1 << '*' << c2 << '=' << Mult(c1, c2) << endl ;
//вивід на консоль результату роботи функції для дійсних змінних
       cout << f1 << '*' << f2 << '=' << Mult(f1, f2) << endl ;
//вивід на консоль результату роботи функції для цілих змінних
       cout << i1 << '*' << i2 << '=' << Mult(i1, i2) << endl ;
}
```

Результатом роботи цієї програми будуть наступні рядки:

```
3 * (-7) = -21
2.3 * 4.1 = 9.43
321 * 2 = 642
```

Таким чином, використання шаблону дозволило функції Mult() працювати з будь-яким із базових числових типів (за прикладом з **char**, **float**, **int**).

Під час використання шаблону, об'єм пам'яті, яку займає програма, не змінюється. Але використання шаблонів приводить до зменшення лістинга програми і спрощенню процесу виправлення помилок.

Під час компіляції програми, компілятор визначає який об'єм пам'яті необхідно виділити шаблонній функції на основі типу даних аргументів, що передаються у цю функцію. Таким чином, шаблон функції не є функцією в звичному розумінні. Це скоріше модель, за якою компілятор, зустрічаючи виклик, формує у пам'яті програмний код.

Шаблони функцій можуть мати як один, так і декілька аргументів, що передаються. Причому рішення про те, скільки із них будуть мати шаблонний тип даних, залишається за програмістом. Значення, що повертається, також може і не бути шаблонного типу.

Приклад шаблона функції для знаходження максимального елемента масиву:

```
template <class A>              //зазначення шаблону типу даних А
//оголошення та визначення шаблонної функції пошуку максимального
//елемента масиву
A GetMaxVal (A *Arr, int n)
{
        int max = *Arr;     //оголошення та визначення за умовчуванням змінної
        for(int i=1;  i < n; i++)   //цикл перебору елементів масиву з 1-го
        {
                if (max < *(Arr+i))         //якщо поточний елемент більший за max
                        max = *(Arr+i);     //перевизначити max
        }
        return max;             //повернути знайдене значення
}
```

З прикладу видно, що ім'я шаблонного типу зовсім не має бути однаковим. Як і для змінних значень, імена шаблонних типів можна вибирати будь-які.

В наведеному прикладі шаблонний тип А використовується для одного із двох аргументів і для значення, що повертається.

Необхідно пам'ятати, що для кожної шаблонної функції шаблонний тип даних з вказаним ім'ям заміщує лише один базовий чисельний. Використовувати шаблонну функцію, як наведено у наступному прикладі, не можна!

Приклад некоректного використання шаблона функції для знаходження добутку двох чисел:

```
template <class B>              //зазначення шаблону типу даних В
//оголошення та визначення шаблонної функції розрахунку добутку
B Mult(B a, B b)
{return  a*b;}                  //повернути значення добутку

void main()                     //оголошення та визначення головної функції
{
```

```
	int  a = 5;			//оголошення та визначення цілої змінної
	float  b = 3.1; //оголошення та визначення змінної з плаваючою точкою
//вивід результату розрахунку на консоль – хибне застосування шаблону
//	cout << a << '*' << b << '=' << Mult(a, b) << endl ;
}
```

Таке використання шаблону функції Mult() недопустимо, оскільки значення a і b мають різні типи даних, а при визначенні шаблону було вказано, що обидва аргументи мають бути однакового типу.

Та все ж, подібна заборона ще не означає, що задача, в якій необхідно виконувати операції зі значеннями різного типу даних, не має рішення. Просто необхідно використовувати одразу декілька шаблонних типів даних.

Приклад створення і використання шаблону функцій для знаходження в масиві елемента найбільш наближеного до значення, що передається:

```
#include <iostream>			//підключення бібліотеки потокового вводу/виводу
using namespace std;		//застосування простору імен std

template <class aT, class bT > //зазначення шаблонів типів даних aT та bT
//оголошення та визначення шаблонної функції пошуку близького елемента
int GetValIndex(aT *Arr, bT val, int n)
{
	int index;			//оголошення цілої змінної для пошуку індексу
	float delta;			//оголошення дійсної змінної для відхилення
//розрахунок відхилення першого елемента від значення
	delta = ((val – *Arr)<0) ? – (val – *Arr) : (val – *Arr);
	index = 0;			//визначення змінної індексу нулем
	for(int i = 1; i < n; i ++) //цикл перебору елементів масиву з 1-го
//якщо відхилення поточного менше за знайдене
		if(delta > (((val – *(Arr+i))< 0) ? –(val – *(Arr+i)) : (val – *(Arr+i))))
		{
//розрахунок відхилення поточного елемента від значення
			delta = ((val – *(Arr+i))<0) ? –(val – *(Arr+i)) : (val – *(Arr+ i));
			index = i;	//визначення змінної індексу поточним і
		}
	return index;			//повернути індекс елемента
}

void main()				//оголошення та визначення головної функції
{
//оголошення та визначення цілого масиву
	int Arr[10] = {1,2,3,4,5,6,7,8,9,0};
//вивід результату виклику шаблонної функції на консоль
	cout << "element N =" << GetValIndex(Arr, 3.721, 10) << endl;
}
```

Функція дозволяє знайти позицію в масиві числа, яке найбільш відповідає значенню 3.721*. Таким числом є* 4*, яке в масиві має індекс* 3*. Функція працює одразу з двома шаблонними типами даних* aT *і* bT.

Результатом роботи програми буде рядок:

element N = 3

Шаблони функцій можуть бути використані і з типами даних користувачів. Однак у цьому випадку необхідно уважно перевіряти оператори, які використовуються в тілі такої функції для виконання дій над даними, що передаються. Адже будь-який оператор може бути неперевантаженим для типу даних користувача або виконувати будь-яку іншу задачу.

15.1.2. Шаблон класу

З шаблонами класів все складається так само, як і з шаблонами функцій. Вони використовуються для розробки моделей класів, які можуть працювати з даними різних типів. Формат оголошення шаблону класу наведено нижче:

```
template  < class <ім'я шаблонного типу даних> >
class  <ім'я класу>
{
     <поля і методи класу>
}
```

Тут, як і для шаблонів функцій, ім'я шаблонного типу даних може бути будь-яким. Шаблон класу може включати декілька шаблонних типів даних.

Взагалі, розробка шаблонів класів переслідує дві цілі: створення засобу для збереження даних будь-якого типу і об'єднання функцій для роботи с такими даними в групу. Ці цілі перекликаються з основною задачею, яка переслідується при створенні будь-якого класу.

При роботі з шаблонами класів, на відміну від шаблонів функцій, існує лише одна відмінність: якщо під час виклику шаблонної функції тип даних визначається компілятором автоматично, то при створенні шаблонного об'єкта класу тип даних необхідно чітко вказати.

Приклад створення і використання шаблону класу для виконання найпростіших математичних операцій над масивом базового числового типу:

```
#include <iostream>       //підключення бібліотеки для потокового вводу/виводу
using namespace std;      //застосування простору імен std

template <class T>        //зазначення шаблону типу даних T
class CMath               //оголошення класу CMath
{
```

```
	public:				//відкриті члени класу
//оголошення та визначення шаблонної функції отримання суми елементів
//масиву
		T  GetSum(T * Arr, int n)
		{
			T  S = 0;	//оголошення змінної для збереження суми значень
			for (int i = 0; i < n; i ++)//цикл перебору елементів масиву
				S += *(Arr + i);	//додавання до суми
			return S;		//повернення отриманої суми
		}
//оголошення та визначення шаблонної функції отримання середнього
//арифметичного значення елементів масиву
		double GetMidVal(T *Arr, int n)
		{
//оголошення змінної для збереження суми значень
			double S = 0;
			for (int i = 0; i < n; i ++)//цикл перебору елементів масиву
				S += *(Arr + i);	//додавання до суми
			return S/n;		//повернення отриманого значення
		}
};
void main()				//оголошення та визначення головної функції
{
//оголошення та визначення масиву цілих чисел
	int iMas[5] = {3, 173, 370, 14, -100};
//оголошення та визначення масиву дійсних чисел
	float fMas[5] = {7.4, 12.72, -7.11, 0.36, -30.8};
	CMath <int> im;			//оголошення шаблонного об'єкта типу int
//вивід на консоль суми елементів масиву цілих чисел
	cout << "Sum of iMas elements is:" << im.GetSum(iMas, 5) <<endl;
//вивід на консоль середнього арифметичного значенння елементів масиву iMas
	cout << "Mid value of iMas elements is:" << im.GetMidVal(iMas, 5) <<endl;
	CMath <float> fm;		//оголошення шаблонного об'єкта типу float
//вивід на консоль суми елементів масиву дійсних чисел
	cout << "Sum of fMas elements is:" << fm.GetSum(fMas, 5) <<endl;
//вивід на консоль середнього арифметичного значення елементів масиву fMas
	cout << "Mid value of fMas elements is:" << fm.GetMidVal(fMas, 5) <<endl;
}
```

Результатом виконання цієї програми будуть наступні рядки:

```
Sum of iMas elements is: 460
Mid value of iMas elements is: 92
Sum of fMas elements is: -17.43
Mid value of fMas elements is: -3.486
```

З прикладу видно, що тип даних, з яким буде використано шаблон класу для формування в пам'яті шаблонного об'єкта визначається під час оголошення об'єкта в кутових дужках між ім'ям класу та ім'ям об'єкта за наступним форматом:

<ім'я класу> < <ім'я типу даних для створення шаблонного об'єкта> > <ім'я об'єкта класу>;

Рядок оголошення шаблонного об'єкта вказує не лише на виділення пам'яті для усіх полів цього об'єкта класу, а також на створення прототипів для всіх методів класу, які використовують шаблонний тип даних.

Коли визначення методів здійснюється за рамками шаблону класу, необхідно вказати шаблонний тип даних для кожної функції і ідентифікувати при цьому клас, що їх включає, як шаблон.

Приклад створення і використання шаблону класу для формування стека і виконання над ним простих операцій:

```
#include <iostream>        //підключення бібліотеки потокового вводу/виводу
using namespace std;       //застосування простору імен std

template <class T>         //зазначення шаблону типу даних T
class Stack                //оголошення класу реалізації стека
{
      private:             //закриті члени класу
            int count;     //оголошення змінної рахування числа елементів в стеку
            int Max;       //оголошення змінної для задання розмірності стека
            T *Arr; //оголошення змінної-покажчика для динамічного массиву
      public:              //відкриті члени класу
//оголошення та визначення конструктора класу
            Stack(int n) : Max(n)  //ініціалізація Max за допомогою аргументу n
            {
//створення динамічного масиву на Max елементів
                  Arr = new T[Max];
                  count = 0;          //визначення змінної – 0 елементів
            }
            bool Push(T val);  //оголошення методу додавання значення у стек
            bool Pop(T& var);//оголошення методу вилучення значення зі стека
            void Clear();       //оголошення методу очищення стека
//оголошення методу отримання кількості елементів стека
            int GetCount();
//оголошення методу отримання максимального значення стека
            bool GetMaxVal(T& max);
//оголошення методу отримання мінімального значення стека
            bool GetMinVal(T& min);
```

```
        ~Stack ();              //оголошення деструктора класу
};

//визначення методу додавання значення у стек
template <class T>                      //зазначення шаблону типу даних T
bool Stack <T>::Push(T val)
{
    if (count != Max)                   //якщо в стеку є вільне місце
    {
        *(Arr + count) = val;           //записати у стек значення
        count++;                    //збільшити значення лічильника елементів
        return true;                    //повернути  true
    }
    else                                //якщо в стеку немає вільного місця
        return false;                   //повернути  false
}

//визначення методу вилучення значення зі стека
template <class T>                      //зазначення шаблону типу даних T
bool Stack <T>::Pop(T& var)
{
    if (count == 0)                     //якщо стек порожній
        return false;                   //повернути  false
    else                                //якщо стек не порожній
    {
        count--;                    //зменшити значення рахунку елементів
        var = *(Arr + count);           //записати в var значення
        return true;                    //повернути  true
    }
}

//визначення методу очищення стека
template <class T>                      //зазначення шаблону типу даних T
void Stack <T>::Clear()
{
    count = 0;                          //визначити лічильник нулем
}

//визначення методу отримання кількості елементів стека
template <class T>                  //зазначення шаблону типу даних T
int Stack <T>::GetCount()
{
    return count;                   //повернути значення лічильника елементів
}
```

```
//визначення методу отримання максимального значення стека
template <class T>                          //зазначення шаблону типу даних T
bool Stack <T>::GetMaxVal(T& max)
{
        if (count == 0)                     //якщо стек порожній
                return false;               //повернути  false
        else                                //якщо стек не порожній
        {
                max = *Arr;  //присвоїти max значення першого елемента стека
//перебрати елементи стека, починаючи з 2-го
                for (int i =1; i < count; i ++)
//якщо значення поточного елемента більше max
                        if (max < *(Arr + i))
                                max = *(Arr + i);  //перевизначити значення max
                return true;                // повернути  true
        }
}

//визначення методу отримання мінімального значення стека
template < class T >                        //зазначення шаблону типу даних T
bool Stack <T>::GetMinVal(T& min)
{
        if (count == 0)                     //якщо стек порожній
                return false;               //повернути  false
        else                                //якщо стек не порожній
        {
                min = *Arr;  //присвоїти min значення першого елемента стека
//перебрати елементи стека, починаючи з 2-го
                for (int i =1; i < count; i ++)
//якщо значення наявного елемента меньше за min
                        if (min > *(Arr + i))
                                min = *(Arr + i);  //перевизначити значення min
                return true;                //повернути true
        }
}

//визначення деструктора класу
template <class T>                          //зазначення шаблону типу даних T
Stack <T> :: ~Stack()
{
        delete [] Arr;                      //видалення динамічного масиву
}
```

```
void main()                     //оголошення та визначення головної функції
{
//оголошення шаблонного об'єкта is цілого типу і створення стека на 10
//елементів
	Stack <int> is(10);
//оголошення шаблонного об'єкта fs дійсного типу і створення стека на 5
//елементів
	Stack <float> fs(5);
//оголошення масиву цілих чисел
	int iMas[15] = {3, 5, 7, -4, 0, -12, 82, 100, -31, 300, 27, 42, -13, -29, -100};
//оголошення масиву дійсних чисел
	float fMas[7] = {3.3, 7.9, -3.4, 15.01, 9.6, 0.17, 10.7};
	int i =0;	//оголошення та визначення змінної для перебору елементів
	while (is.Push (iMas [i++]))		//цикл заповнення стека is
	{ }
	i = 0;				//визначення змінної перебору елементів нулем
	while (fs.Push (fMas [i++]))		//цикл заповнення стека fs
	{ }
	int v;  //оголошення змінної цілого типу для збереження значення зі стека
	if (is.Pop (v))		//якщо стек не порожній, то зчитати значення в v
//вивід зчитаного значення на консоль
		cout << "INT Stack last value was: " << v << endl;
	if (is.Pop (v))		//якщо стек не порожній, то зчитати значення в v
//вивід зчитаного значення на консоль
		cout << "INT Stack last value was: " << v << endl;
//вивід на консоль числа елементів  у стеку цілого типу
	cout << "In INT Stack are " << is.GetCount () << "values" << endl;
//вивід на консоль числа елементів  у стеку дійсного типу
	cout << "In FLOAT Stack are " << fs.GetCount () << "values" << endl;
//оголошення змінної цілого типу для максимального значення стека is
	int imax;
//якщо стек не порожній, то отримати в imax максимальне значення
	if (is.GetMaxVal (imax))
//вивід на консоль максимального значення стека is
		cout << "Max value of INT Stack is: " << imax << endl;
//оголошення змінної дійсного типу для мінімального значення стека fs
	float fmin;
//якщо стек не порожній, то отримати в fmin мінімальне значення
	if (fs.GetMinVal (fmin))
//вивід на консоль мінімального значення стека fs
		cout << "Min value of FLOAT Stack is: " << fmin << endl;
}
```

Результатом виконання даної програми будуть наступні рядки:

```
INT Stack last value was: 300
INT Stack last value was: -31
In INT Stack are 8 values
In FLOAT Stack are 5 values
Max value of INT Stack is: 100
Min value of FLOAT Stack is: -3.4
```

Перші два рядки характеризують коректність роботи стека. Виклик в циклі метода Push() приводить до перезапису із iMas в стек типу цілих чисел 10 значень (по 300), а із fMas в стек типу дійсних чисел 5 значень (по 9.6). Потім, два виклики метода Pop() для стека типу цілих чисел вилучають два останніх значення (300 та -31).

Третій та четвертий рядок вказують скільки значень в кожному із стеків (8 в типі цілих чисел і 5 в типі дійсних чисел). Останні рядки виводять на консоль для стека типу цілих чисел максимальне, а для стека типу дійсних чисел – мінімальне значення.

Як видно з прикладу, для того, щоб показати, що функція є методом шаблона класу, необхідно дотримуватись наступного формату:

template < **class**<*ім'я шаблонного типа даних*>>
<*тип даних*> <*ім'я шаблону класу*> <<*ім'я шаблонного типу даних*>> :: <*ім'я функції*> (<*список аргументів*>)
{
<*тіло функції*>
}

В основному шаблони класів використовуються для створення так званих контейнерів для збереження даних (складно організованих масивів). Для мови C++ спеціально розроблена допоміжна бібліотека STL (Standart Template Library – стандартна бібліотека шаблонів), яка містить основні види контейнерів та засоби роботи з ними.

Під час створення шаблонів функцій і класів бажано виконувати спочатку розробку простої функції або класу, що базуються на певному типі даних, а потім, після їх виправлення та тестування, виконувати перетворення в шаблон.

Висновки

У даному розділи було розглянуто наступні основні питання:
– застосування шаблонів типів класів;
– шаблони функцій;
– шаблони класів.

Контрольні питання

1) Що називають шаблоном?
2) Для чого застосовуються шаблони?

3) За допомогою якого службового слова програми створюється шаблон типу даних?

4) Які шаблони є найбільш поширеними?

5) Наведіть формат оголошення шаблонної функції.

6) Яке службове слово замість **class** можна використовувати для створення шаблонного типу?

7) Як називають виклик шаблонної функції?

8) У чому різниця між звичайними та шаблонними функціями відносно компілятора?

9) Скільки аргументів може мати шаблонна функція?

10) Чому не можна застосовувати шаблонну функцію з одним шаблонним типом для передачі двох різнотипних аргументів?

11) Чому треба бути обережним при застосуванні шаблонів для складених типів даних?

12) Наведіть формат створення шаблону класу.

13) Як визначаються методи шаблонного класу поза його оголошення?

14) Яка бібліотека містить основні види контейнерів та засоби роботи з ними?

15) Для чого використовуються шаблони?

16. БАГАТОФАЙЛОВІ ПРОЕКТИ

Навчальною метою розділу є ознайомлення читача зі створенням великих програмних проектів, що включають декілька файлів з кодом.

Внаслідок вивчення матеріалу даного розділу читач повинен вміти:

– виділяти оголошення програмного коду у заголовні, а визначення у програмні файли;

– поєднувати декілька файлів з програмним кодом в один проект;

– реалізовувати та застосовувати міжфайлові змінні, функції та класи.

16.1. Багатофайлові проекти

При написанні великих програм їх код може займати декілька сотень сторінок. Зрозуміло, що розміщувати такі програми в одному файлі незручно. Редагування та відлагодження такого коду потребували б значних витрат додаткового часу, який майже завжди є чітко обмеженим при створенні програмного забезпечення. Тому майже всі мови програмування, у тому числі C++, мають механізми для формування багатофайлових проектів, кожен з файлів котрих включає структурну частину коду, що значно спрощує написання програм.

У мові C++ розрізняють дві групи файлів: заголовні (розширення ".h") та програмні (розширення ".c" та ".cpp"). У заголовних файлах розміщують оголошення, а у програмних визначення. Підключення заголовних файлів було розглянуто на початку ознайомлення з основами побудови програм.

Воно виконується за допомогою директиви #**include**. Але створення таких файлів раніше не розглядалося. Компілятор сприймає весь код, що записано у заголовний файл як частину тієї програми, де виконано директиву підключення. Таким чином, ніяких особливостей при створенні файлів бібліотек немає. Треба лише зрозуміти, що все, що було оголошено у таких файлах набуває глобальної видимості, бо додається у точці застосування відповідної директиви.

Наприклад, програма, що працює з методами класу для простих обчислень, може мати наступний вигляд:

mclass.h

```
class Cmclass                      //оголошення класу математичних функцій
{
      public:                      //відкриті члени класу
//оголошення та визначення методу розрахунку суми
            float sum(float a, float b)
            {
                  return a+b;      //повернути результат суми змінних
            }
//оголошення та визначення методу розрахунку різниці
            float sub(float a, float b)
            {
                  return a-b;      //повернути результат різниці змінних
```

```
        }
//оголошення та визначення методу розрахунку добутку
        float mult(float a, float b)
        {
            return a*b;         //повернути добуток змінних
        }
//оголошення та визначення методу розрахунку ділення
        float div(float a, float b, bool &r)
        {
            if(b!=0)            //якщо у знаменнику не 0
            {
                r=1;    //визначити прапорець за допомогою значення 1
                return a/b; //повернути результат ділення змінних
            }
            else                //якщо у знаменнику 0
            {
                r=0;    //визначити прапорець за допомогою значення 0
                return 0;   //повернути 0
            }
        }
}mc;                            //оголошення об'єкта класу
```

main.cpp

```
#include <iostream>             //підключення бібліотеки вводу/виводу
#include <iomanip>              //підключення бібліотеки маніпуляторів
#include "mclass.h"             //підключення заголовного файлу

using namespace std;            //застосування простору імен

void main()                     //оголошення та визначення головної функції
{
    float rez;                  //оголошення змінної з плаваючою точкою
    bool f;                     //оголошення логічної змінної
    float x,y;                  //оголошення змінних з плаваючою точкою

    char c;                     //оголошення символьної змінної
    cout<<"Calculating//"<<endl; //вивід константного рядка
    for(;;)                     //вічний цикл
    {
//вивід константного рядка
        cout<<"Mult-0;Div-1;Sum-2;Sub-3"<<endl;
        cout<<"Make your choice:"<<flush; //вивід запиту до користувача
        cin>>c;                 //отримання значення від користувача
        cout<<"Enter x:"<<flush;    //вивід запиту до користувача
        cin>>x;                 //отримання значення від користувача
```

```
            cout<<"Enter y:"<<flush;        //вивід запиту до користувача
            cin>>y;                         //отримання значення від користувача
            cout<<"Result is:"<<flush;      //вивід запиту до користувача
            switch(c)                       //перевірка значення змінної
            {
                case '0':                   //якщо введено 0
//виклик функції розрахунку добутку і вивід результату на консоль
                    cout<<mc.mult(x, y);
                    break;                  //вийти з оператора
                case '1':                   //якщо введено 0
//виклик функції розрахунку ділення
                    rez=mc.div(x, y, f);
//вивід константного рядка
                    if(f)                   //якщо у змінній 1
                        cout<<rez; //вивід на консоль результату
                    else                    //інакше
//вивід константного рядка
                        cout<<"divizion by zero";
                    break;          //вийти з оператора
                case '2':                   //якщо введено 0
//виклик функції розрахунку суми і вивід результату на консоль
                    cout<<mc.sum(x, y);
                    break;          //вийти з оператора
                case '3':                   //якщо введено 0
//виклик функції розрахунку різниці і вивід результату на консоль
                    cout<<mc.sub(x, y);
                    break;          //вийти з оператора
                default:                    //якщо інший символ
                    cout<<"Wrong choice";
            }
//вивід запиту до користувача
            cout<<endl<<"Do you want to continue?(Y/N):"<<flush;
            cin>>c;                         //отримання значення від користувача
            if(c=='N')                      //якщо користувач ввів символ 'N'
                break;                      //вийти з оператора
        }
}
```

Програма складається з двох файлів mclass.h *та* main.cpp*. У* mclass.h *виконано оголошення класу* Cmclass *з методами для розрахунку суми, різниці, добутку та частки. Також оголошено об'єкт* mc.

У main.cpp файл mclass.h підключено через подвійні лапки, що вказує компілятору, що його знаходження відповідає шляху до поточного файлу. Головна функція включає вічний цикл, що через діалоговий інтерфейс з

користувачем застосовує через об'єкт mc методи для реалізації математичних операцій.

Робота з одним файлом програми та будь-якою кількістю заголовних є легкою. Більш складним є застосування декількох файлів програм при використанні загальних даних чи функцій. Підключення в кожному однієї бібліотеки не є достатнім. Окрім того, такий підхід може призвести до помилок, бо у програмі буде оголошено декілька змінних з однаковими іменами.

В цьому випадку необхідним є застосування міжфайлових змінних, функцій, класів тощо.

16.2. Міжфайлові змінні

При одночасному оголошенні глобальної змінної val у файлах A.cpp та B.cpp компілятор зреагує на це, як на помилку.

A.cpp

```
int val;              //оголошення глобальної цілої змінної
```

B.cpp

```
int val;              //оголошення глобальної цілої змінної
```

Це пов'язано з тим, що при компіляції буде зустрінуто оголошення змінної val два рази.

Але оголошення змінної в одному з файлів і спроба її використання у інших також призведе до помилки.

A.cpp

```
int val;              //оголошення глобальної цілої змінної
```

B.cpp

```
val=35;               //визначення глобальної змінної
```

У цьому випадку компілятор вважає, що для файла B.cpp оголошення val відсутня.

Задовольнити вимогливість компілятора можна, застосувавши службове слово **extern** за наступним форматом:

extern <*тип даних* > <*ім'я змінної* >;

Це є оголошенням змінної без визначення, яке вказує на те, що повне оголошення знаходиться у іншому файлі.

Завдяки такому підходу програмісту для застосування глобальної міжфайлової змінної є достатнім виконати одне повне її оголошення і оголошення без визначення у всіх файлах, де вона буде задіяна.

A.cpp

```
int val;              //оголошення глобальної цілої змінної
val=35;               //визначення глобальної змінної
```

B.cpp

```
int n;              //оголошення глобальної цілої змінної
extern int val;     //оголошення без визначення глобальної змінної
n=val;              //визначення змінної
```

Але використання **extern** вирішує лише одну з проблем, що виникають у багатофайлових проектах – єдина глобальна змінна для програми. Інша проблема може з'явитися коли є необхідність створити для кожного з файлів свою окрему глобальну змінну. Найпростішим вирішенням є застосування різних імен. Але існує також можливість відокремити змінні з однаковими іменами. Для цього слугує службове слово **static**:

static <*тип даних* > <*ім'я змінної* >;

Це слово говорить компілятору, що оголошення глобальної змінної було виконано лише для поточного файлу.

A.cpp

```
static int val;          //оголошення статичної глобальної цілої змінної
val=47;                  //визначення глобальної змінної
```

B.cpp

```
static int val;          //оголошення статичної глобальної цілої змінної
val=23;                  //визначення глобальної змінної
```

Такого ж змісту набуває застосування службового слова **const**.

A.cpp

```
//оголошення та визначення константної глобальної цілої змінної
const int a=243;
```

B.cpp

```
//оголошення та визначення константної глобальної цілої змінної
const int a=31;
```

Константні глобальні змінні за умовчуванням оголошуються лише для поточного файлу. Але, як і звичайні змінні, вони можуть бути зазначені міжфайловими за допомогою **extern**.

A.cpp

```
//оголошення та визначення константної дійсної глобальної змінної
const float pi=3.14;
```

B.cpp

```
//оголошення та визначення константноїдійсної глобальної змінної
extern const float pi;
float corn=pi/2;         //оголошення та визначення дійсної змінної
```

16.3. Міжфайлові функції

Для того, щоб глобальна функція стала міжфайловою немає необхідності у застосуванні додаткових службових слів. Достатньо у всіх файлах, де її буде використано, виконати оголошення, навіть зі скороченим списком аргументів. При цьому важливо, щоб у всій програмі існувало лише єдине визначення такої функції.

A.cpp

```
//оголошення та визначення глобальної функції розрахунку суми
int Sum(int a, int b)
{
	return a+b;		//повернення результату розрахунку суми
}
```

B.cpp

```
int Sum(int, int);		//оголошення глобальної функції розрахунку суми
//оголошення та визначення цілої змінної значенням, що повертає функція
int rez=Sum(5,12);
```

Такий підхід дозволяє визначити для компілятора як існування глобальної функції, так і кількість пам'яті, що задіяно для зберігання її параметрів.

Для відокремлення глобальних функцій з однаковими іменами використовується вже відоме за міжфайловими змінними службове слово **static**.

A.cpp

```
//оголошення та визначення статичної глобальної функції розрахунку суми
static int func(int a, int b)
{
	return a+b;			//повернення результату розрахунку суми
}
```

B.cpp

```
//оголошення та визначення глобальної статичної функції розрахунку
static int func(int a, int b)
{
	return a*a+b;
}
```

У цьому випадку кожна з функцій повинна мати своє визначення.

16.4. Міжфайлові класи

На відміну від змінних та функцій застосування різних класів з однаковими іменами не має ніякого сенсу.

Тому питання лише у тому, як застосувати один клас у декількох файлах одного проекту. Це здійснюється без застосування додаткових службових слів. Достатньо мати у файлі оголошення класу і його членів для кожного файлу і

лише єдине визначення методів та програму. Як і для міжфайлових функцій, при використанні міжфайлових класів для компілятора є важливим зазначити ім'я і кількість пам'яті.

A.cpp

```
class CMyClass                    //оголошення класу
{
      int var;                    //оголошення цілого поля класу
      CMyClass():var(10);         //оголошення та визначення конструктора
      {
      }
      int func(int a, int b)      //оголошення та визначення методу
      {
            return a*a+b*b+b*a;   //повернення результату розрахунку
      }
};
```

B.cpp

```
class CMyClass                    //оголошення класу
{
      int var;                    //оголошення цілого поля класу
      CMyClass();                 //оголошення конструктора
      int func(int, int);         //оголошення методу
};
```

16.5. Багатофайлові програми

Таким чином, застосування механізму структуризації програми по файлами є зручним і дуже важливим засобом для створення коректних та оптимізованих програм.

Найчастіше кожен клас, що застосовується в об'єктно-орієнтованому проекті, розміщується в двох файлах, заголовному та програмному. Такий підхід надає можливість програмісту окремо працювати зі структурними об'єктами та відлагоджувати їх.

Крім того, розроблені за таким принципом класи можна використовувати багаторазово, виконуючи копіювання їх файлів у новий проект.

Завдяки багатофайловості програма в цілому набуває модульності, що значно полегшує компонування та декомпозицію.

Висновки

У даному розділі було розглянуто наступні основні питання:

– розробка багатофайлового проекту;
– міжфайлові змінні;
– міжфайлові функції;
– міжфайлові класі.

Контрольні питання

1) У чому полягає сутність розташування програмного коду у декількох пов'язаних файлах?

2) Файли з якими розширеннями можуть входити до проекту на мові С++?

3) Яка директива препроцесору дозволяє підключати заголовкові файли?

4) За яким принципом програмний код поділяється між програмними та заголовними файлами?

5) Які змінні називають міжфайловими?

6) Чому у двох програмних файлах проекту не можна виконати одночасне оголошення глобальних змінних з однаковими іменами?

7) Чому глобальна змінна не може бути оголошена в одному програмному файлі проекту, а визначена в іншому?

8) Як глобальну змінну зробити міжфайловою?

9) До чого приводить оголошення глобальної змінної за допомогою службового слова **static**?

10) Як константну змінну зробити міжфайловою?

11) Як створити міжфайлову функцію?

12) Скільки оголошень та скільки визначень потребує міжфайлова глобальна функція?

13) Як глобальні функції з однаковими іменами, що оголошені у різних програмних файлах проекту, зробити незалежними?

14) Як оголосити міжфайловий клас?

15) Чому для міжфайлового класу необхідно зазначувати оголошення у всіх програмних файлах, що містять код роботи з ним?

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

Застосування у сукупності з навчальним посібником наступної літератури дозволить значно розширити та поглибити знання та вміння щодо програмування мовою С++.

1. Лафоре, Р. Объектно-ориентированное программирование в С++. Класика Computer Science [Текст]: пер. с англ./ Р. Лафоре. – 4-е изд. – СПб.: Питер, 2005. – 924 с.: ил.
2. Шилдт, Г. Полный справочник по С [Текст]: пер. с англ./ Г. Шилдт. – 4-е изд. – М.: Издательский дом "Вильямс", 2004 – 704 с.: ил.
3. Шилдт, Г. Полный справочник по С++ [Текст]: пер. с англ./ Г. Шилдт. – 4-е изд. – М.: Издательский дом "Вильямс", 2006 – 800 с.: ил.
4. Шилдт, Г. С++. Руководство для начинающих [Текст]: пер. с англ./ Г. Шилдт. – М.: Издательский дом "Вильямс", 2006 – 672 с.: ил.
5. Страуструп, Б. Язык программирования С++. Специальное издание [Текст]: пер. с англ./ Б. Страуструп – М.: "Бином"-"Невский диалект", 2012 – 1136 с.: ил.
6. Березин, В.И. Начальный курс С и С++ [Текст]/ В.И. Березин, С.Б. Березин – М.: Диалог-Мифы, 2003. – 288 с.
7. Архангельский, А. Я. Программирование в С++ Builder 6 и 2006 [Текст] А.Я. Архангельский, М. А.Тагин – М.: Бином-Пресс, 2007. – 1184с.
8. Седжвик, Р. Фундаментальные алгоритмы на С. Анализ/Структуры данных/Сортировка/Поиск [Текст]: пер. с англ./ Р. Седжвик – СПб: ООО "ДиаСофт", 2003. – 672 с.
9. Солтер, Николас А. С++ для профессионалов. [Текст]: пер. с англ./ Николас А. Солтер, Скотт Дж. Клепер – М.: Издательский дом "Вильямс", 2006. – 912 с.: ил.

ПЕРЕЛІК СКОРОЧЕНЬ

АКС – Автоматизація комп'ютерних систем

ANSI – American National Standards Institute – Національний інститут стандартизації США

ASCII – American Standard Code for Information Interchange – Американський стандартний код для інформаційного обміну

BIN – Binary – Двійкова система числення

C89 – Стандарт ANSI/ISO мови C 1989 року

C99 – Стандарт ANSI/ISO мови C 1999 року

DEC – Decimal – Десяткова система числення

FIFO – first in, first out – Обслуговування черги, що засновано на правилі – першим з'явився, першим обслуговується

HEX – Hexadecimal – Шістнадцяткова система числення

ISO – International Standards Organization – Міжнародна організація по стандартизації

LIFO – last in, first out – Обслуговування черги засноване на правилі – останнім з'явився, першим обслуговується

OCT – Octal – Вісімкова система числення

ПРЕДМЕТНИЙ ПОКАЖЧИК

A
ANSI 8
ASCII 18, 42, 52, 55
asm 17
auto 17
automatic 91

B
bool 17, 37
break 17, 37, 45, 56

C
C89 8, 17
C99 8, 12
case 17, 45
catch 17
char 17, 25
class 17, 110
const 17, 19, 164
continue 17, 37, 57

D
default 17, 45
delete 17, 63, 73
do-while 17, 37, 50, 52
double 17, 25

E
enum 17, 80
extern 17, 163

F
FIFO 74, 97
float 17, 25
for 17, 37, 50, 54
friend 17, 140

G
goto 17, 37, 57

I
if-else 17, 36, 48
include 14
inline 17
int 17
ISO 8

L
LIFO 74, 101
long 17, 25

N
namespace 108
new 17, 63, 73

O
operator 17

P
private 17, 111, 120
protected 17, 111, 120
public 17, 111, 120

R
register 17
return 17, 84

S
short 17, 25
signed 17, 25
sizeof 17
static 17, 91, 143, 164
struct 17, 76, 86
switch 17, 36, 45

T
template 17, 148
this 17
throw 17
try 17
typedef 17

U
union 17, 79
unsigned 17
using 108

V
virtual 17, 134
void 14, 62
volatile 17

W
while 17, 37

А
абстрактний 138
автоматичний 68, 91
адреса 32, 61, 96
аргумент 63, 83, 87, 90, 92

Б
базовий клас 120, 123, 134, 140
байт 18, 61, 71, 79, 107
бібліотека 8, 13, 86, 107, 158, 160
біт 18, 25
блок 10, 13, 26, 37, 45, 47, 50, 53, 55
буфер 51, 65, 74, 97, 101

В
ввід/вивід 29, 106
вектор 65
визначення:
– змінної 19
– функції 13, 84
виклик 84
вираз 16, 19, 25, 37, 46, 48, 51, 53
віртуальний 138, 140

Г
глобальний 11, 26

Д
декремент 20
деструктор 113, 124, 138
директива 15
доступ 11, 18, 61, 63, 68, 77, 80, 100, 111, 119
дружній 140

Е
екземпляр 76
елемент 16, 61, 63, 65

З
заголовний 15, 160
змінна 18, 61
змінна-покажчик 18, 61, 71, 78, 87, 110, 134
зсув 22

І
ім'я:
– змінної 18, 61
– масиву 65
– мітки 58
– перелічення 80
– поєднання 79
– структури 76
– функції 83
індекс 65
ініціалізація 19
інкремент 20
інтерпретатор 11

К
клас 113, 120, 123, 134, 140, 152
коментування 16
компілятор 9, 11, 13
константа 19
конструкція 43
конструктор 113, 124

Л
локальна 26

М
маніпулятор 106
масив 65, 72
матриця 65, 71
метод класу 111
мова 8
модифікатор 31

О
об'єкт 110
оголошення 18, 61, 66, 83, 111

оператор 20, 36, 45, 57, 61
операція 20, 48, 61, 112

П

пам'ять 61, 63
перевантаження 92
перечислення 80
поєднання 79
поле 111
поліморфізм 134
посилання 87
потік 106
похідний клас 119
препроцесор 13
присвоювання 24
пріоритет 25

Р

рекурсія 93
розгалуження 36
розмірність 66

С

спадкування 119
специфікатор:
– формату 30, 33
– доступу 111, 120
список 96
співвідношення 23
статичний 142
стек 74, 101, 154
структура 86
схема 37, 45, 50, 53, 55

Т

таблиця:
– істинності 21, 22
– маніпуляторів 106
– операторів 20
– пріоритетів 25
– специфікаторів 30, 33
тип 17, 76
тіло 13, 37, 51, 53, 54

Ф

функція 83, 134, 140, 142, 148

Ц

цикл 50

Ш

шаблон 148, 152

ДЛЯ НОТАТОК

ДЛЯ НОТАТОК

ДЛЯ НОТАТОК

Навчальне видання

Ткачов Віктор Васильович
Огєєнко Павло Юрійович
Макітренко Роман Васильович

КОМП'ЮТЕРНІ ТЕХНОЛОГІЇ ТА ПРОГРАМУВАННЯ

Навчальний посібник

Том 1
ТЕОРЕТИЧНІ ВІДОМОСТІ

Редактор Є.М. Ільченко

Підписано до друку 19.06.2012 р. Формат 30x42/4
Папір офсет. Ризографія. Ум. друк. арк. 10,2
Обл.-вид. арк. 14,2. Тираж 100 пр. Зам. №

Підготовлено до друку та видрукувано
у Державному ВНЗ "Національний гірничий університет"
Свідоцтво про внесення до Державного реєстру ДК № 1842
від 11.06.2004 р.

49005, м. Дніпропетровськ, просп. К.Маркса, 19